DONALD F. ZIMMER, JR. (SBN 112279)
fzimmer@kslaw.com
CHERYL A. SABNIS (SBN 224323)
csabnis@kslaw.com
KING & SPALDING LLP
101 Second Street - Suite 2300
San Francisco, CA 94105
Telephone: (415) 318-1200
Facsimile: (415) 318-1300

SCOTT T. WEINGAERTNER (*Pro Hac Vice)*
sweingaertner@kslaw.com
ROBERT F. PERRY
rperry@kslaw.com
BRUCE W. BABER (*Pro Hac Vice)*
bbaber@kslaw.com
KING & SPALDING LLP
1185 Avenue of the Americas
New York, NY 10036-4003
Telephone: (212) 556-2100
Facsimile: (212) 556-2222

IAN C. BALLON (SBN 141819)
ballon@gtlaw.com
HEATHER MEEKER (SBN 172148)
meekerh@gtlaw.com
GREENBERG TRAURIG, LLP
1900 University Avenue
East Palo Alto, CA 94303
Telephone: (650) 328-8500
Facsimile: (650) 328-8508

Attorneys for Defendant
GOOGLE INC.

UNITED STATES DISTRICT COURT

NORTHERN DISTRICT OF CALIFORNIA

SAN FRANCISCO DIVISION

ORACLE AMERICA, INC.

Plaintiff,

v.

GOOGLE INC.

Defendant.

Case No. 3:10-cv-03561-WHA

SUPPLEMENTAL DECLARATION OF TRUMAN FENTON

Dept.: Courtroom 9, 19th Floor

Judge: Honorable William Alsup

Tutorial: April 6, 2011, 1:30 p.m.

Hearing: April 20, 2011, 1:30 p.m.

I, Truman Fenton, hereby declare and state as follows:

1. I am an attorney with the law firm of King & Spalding LLP, which is counsel of record for Google Inc. I have personal knowledge of the facts set forth in this declaration unless otherwise noted, and, if called to do so, I could and would competently testify thereto.

2. Exhibit G is a true and correct copy of John Gough, *Virtual Machines, Managed Code and Component Technology*, at 1-2 (IEEE 2005) obtained from the CiteSeerX scholarly document repository hosted by Pennsylvania State University.

3. Exhibit H is a true and correct copy of John Aycock, *A Brief History of Just-In-Time*, 35 ACM Computing Survs. 97 (June 2003) obtained from the Association for Computing Machinery website.

4. Exhibit I is a true and correct copy of excerpts of Plaintiff's Responses and Objections to Defendant Google Inc.'s First Set of Interrogatories to Plaintiff Oracle America, Inc. (Nos. 1-10).

5. Exhibit J is a true and correct copy of Steve Lohr, *Software War Pits Oracle vs. Google*, N.Y. Times, August 30, 2010, obtained from Westlaw.

6. Exhibit K is a true and correct copy of FoxBusiness.com, "Google Android, Video Games Dominate Mobile World Congress," http://www.foxbusiness.com/technology/2011/02/21/google-android-video-games-dominate-mobile-world-congress/ (Feb. 21, 2011) obtained from the FoxBusiness.com website at on Mar. 31, 2011.

7. Exhibit L is a true and correct copy of WSJ.com, "Mobile World Congress: Google's Android Big in Barcelona," http://online.wsj.com/search (Feb. 17, 2011) obtained from the WSJ.com website on March 31, 2011 by navigating to the WSJ.com website, entering "Google Android Barcelona" in "Search for" box; clicking "Search"; and selecting the link entitled "Mobile World Congress: Android Big in Barcelona."

8. Exhibit M is a true and correct copy of a post on Jonathan Schwartz's Blog entitled "Congratulations Google, Red Hat and the Java Community" that is dated Nov. 5, 2007. This document was obtained from http://blogs.sun.com/jonathan/entry/congratulations_google on Mar. 31, 2011.

9. Exhibit N is a true and correct copy of excerpts of J. Gosling et al., *The NeWS Book* at 19 (Springer-Verlag 1989).

10. Exhibit O is a true and correct copy of a newsgroup message posted by James Gosling entitled "Re: Eolas acquires milestone internet software patent" and dated Aug. 21, 1995. This document was obtained from the World Wide Web Consortium (W3C) website on November 11, 2010.

11. Exhibit P is a true and correct copy of excerpts from the Microsoft Press Computer Dictionary 374 (2d ed. 1994).

12. Exhibit Q is a true and correct copy of excerpts from David Gries, *Compiler Construction for Digital Computers* at 244-45 (John Wiley & Sons, Inc. 1971).

I declare under penalty of perjury under the laws of the United States of America that the foregoing is true and correct and that this declaration was executed this 31th day of March, 2011, in Austin, Texas.

Dated: March 31, 2011

Truman Fenton

EXHIBIT G

Virtual Machines, Managed Code and Component Technology

John Gough
Queensland University of Technology
Brisbane, Australia

Abstract

Abstract machines have been used as an implementation mechanism for programming languages for more than thirty years. In their latest incarnation execution engines based on virtual machines offer "Managed Execution". The implications of this change go far beyond the superficial advantages of platform portability and go to the heart of software reliability.

In this paper it is argued that managed execution platforms such as the .NET Common Language Runtime and the Java Virtual Machine form the only reasonable basis for trustworthy component software. There is also an overview of current research in this field, including the vexed question of version evolution.

1. Introduction – A Brief History

The definition of abstract machines as a mechanism for *reasoning* about programs goes back to the dawn of computer science as we now understand it. The use of such machines to define program representation for compilers dates back at least to Wirth's "P-machine" in 1973[1]. The P-machine was a stack-based abstract machine intended to define an intermediate form for the "portable" *Pascal* compiler. Porting the compiler to a new execution platform required the creation of a new "back-end" that transformed the instructions of the abstract machine into the binary code of the target machine. However, a standard part of the porting process was to write an *interpreter* to simulate the P-machine on the new target, as a first step in the bootstrap process. This particular strategy had an unforseen consequence when Kenneth Bowles at *UCSD*[2] dispensed with the back-end and wrote tiny interpreters for the "*P-Code*" hosted on the multitude of incompatible microprocessors that were appearing in the early 1970s. The use of an abstract machine in this context was pivotal because of the simplicity of the interpreter that was needed. A further benefit was the extremely high code density that *P-Code* achieved — a very important factor in the days of tiny memories.

The P-machine was designed to support just one programming language, although it was expressive enough to support some other languages also. A later development of the concept was *U-Code*[3] which was used as an intermediate form for a number of mainstream commercial compilers for several languages on many target machines. With *U-Code*, as with *P-Code*, the operational semantics of the instructions are defined by an abstract stack-based automaton. It may be argued that the main role of the abstract machine at this time was one of *compiler factorization*. By using the common intermediate form, the problem of compiling N different languages for M different machines is reduced to producing N "front-ends" and M "back-ends" rather than $N \times M$ monolithic compilers.

A typical expression evaluation in such a system would be fetching the value of a 8-byte floating point field b of some structure a. Typical code for a stack machine of the era would be—

```
ldadr 'a'   // load address of 'a'
ldc.i4 4    // offset of field 'b'
padd        // pointer arithmetic
deref.r8    // load 8-byte real
```

Several things are worth noting at this stage: the front-end apparently needs to know the layout of the structure on the target machine, and the language is *untyped* except for the integer/floating point separation. Although front-ends based on this scheme produced "portable" code, the output was generally parameterized for each target machine. My own *Gardens Point* compilers were typical[4]. They required to know the following target information: alignment rules, argument assembly conventions, stack mark size and which of three possible methods were used for passing structures by value.

Abstract machines have also played an important part in the implementation of specialist languages. The *Warren Abstract Machine*[5] for the *Prolog* language is a typical example. There are at least two factors at work here. Languages that have dynamic aspects that make static compilation unattractive have routinely used interpretation for their implementation. The interpreter emulates an abstract ma-

Proceedings of the 2005 Australian Software Engineering Conference (ASWEC'05)
1530-0803/05 $20.00 © 2005 **IEEE**

chine that bridges the semantic gap between the source language and the instructions of the host machines. Furthermore, the use of an intermediate language factors the implementation problem when multiple machine architectures are targetted.

So far, according to this brief history, abstract machines have been used in two ways. A stream of instructions for an abstract machine may be used, transiently, as an intermediate representation for program texts inside factored compilers. Alternatively, the instructions for the abstract machine have been the end product of the compiler, to be subsequently executed by an interpreter when the program is run.

In 1989 a further possibility became apparent. In that year the Open Software Foundation called for proposals for an *Architecture Neutral Distribution Form* (*ANDF*) for computer programs[6]. The idea was that programs written in any source language would be compiled into an architecture neutral form, supported by the facilities of a standard environment. Each such program could be run on any machine that possessed an "*Installer*" utility. The installer would complete the compilation process by transforming the *ANDF* into native code for the particular computer.

It is relevant to note that the demands of *ANDF* necessitate a higher level of abstraction than is the case for *U-Code* and similar abstract machine forms. In this case there can be no "parameterization of output for the target machine" since the target is unknown at compilation time. Such matters as the layout of structures needs to be deferrred for the installer to decide. The code of our previous code snippet needs to contain *symbolic information*. Something along the following lines is needed, where we have arbitrarily assumed that the type of *a* is *AType* from module *Mod*—

```
ldadr Mod.AType::'a'          // load adr of 'a'
ldfld real64 Mod.AType::'b' // load field value
```

Consider the second instruction. Field names need to be qualified with the type of the field *and* the type of the structure to which the field belongs. Furthermore, the distribution form must contain type declaration *metadata* sufficient to allow the installer to lay out the data.

ANDF was not a success in the market, although most compiler people agreed that it was a stunningly good idea. At least part of the problem was the very high levels of symbolic information that *ANDF* contained. Major software vendors tended to be spooked by the fact that anyone with an *ANDF* distribution could reconstruct fully typed, readable source code with minimal effort.

The next significant step in our saga arose from the doctoral research of Michael Franz[7], a student at *ETH*, Zurich. This work was continued later at UCI. Michael gave a final twist to the installer idea by invoking the installer each time the program was loaded, rather than just once when the progam was installed on the machine. Nowadays, we would call the "installer" a just-in-time compiler (*JIT*). The "slim binaries" that were the distribution format were for a single language (*Oberon-2*) but were used unchanged on different architectures. The distribution form was extremely compact, despite the necessary presence of the metadata. The mind-blowing result from the research was that the time saved in reading the smaller binary from the disk more than compensated for the processor time to perform code generation within the installer.

And then in May 1995 language *Java* was announced by *Sun Microsystems*[9]. The distribution format of *Java* is part of the definition of the language, and is based on an abstract machine: the Java virtual machine (*JVM*)[8]. Java was always intended to be executed either by interpretation or by *JIT* compilation, and still is. The output of the *Java* compiler is one or more "*class*" files for every *Java* source file. Each such class file contains the "bytecodes" that are the instructions for the stack-based virtual machine together with the metadata that is necessary to allow true target independence. The new element that *Java* added to the abstract machine story was *verification*. Since all user-declared data is statically typed in every class file it is possible to use a lightweight "theorem prover" to check that the code is type-safe and hence memory-safe. Now, every legal *Java* program is necessarily type-safe so it may appear to be overkill for the *JIT* to re-check what the compiler has already guaranteed. This would certainly be the case if the class-files were only a transient intermediate representation between compiler phases. However, the generation of the class files and the invocation of the *JIT* are separated both temporally and spatially. The browser that downloads a "*Java*" class file as a component of an applet cannot trust that the bytecodes were generated by a correct compiler, nor that the code has not been modified either accidentally or with evil intent.

The *JVM* was designed with the goal of supporting just one language: *Java*. Nevertheless with more or less difficulty the *JVM* can support (type-safe subsets) of an alarmingly long list of languages.

The final event in our brief history took place in mid-2000, when *Microsoft* announced their *.NET* system. This system is supported by the Common Language Runtime (*CLR*), another stack-based abstract machine. It features a more expressive type system than *Java*, and is explicitly designed to support a wide range of languages — including those that are type-safe *and* those that are not. The type-safe ones can be verified by the *JIT* while the the type-unsafe ones skate on the same thin ice as any other binary program representation. The authoritative source on the *Common Language Infrastructure* which includes the *CLR*, is the annotated standard[10].

Generically we refer to the *CLR* and the *JVM* are being *managed execution systems*. They are managed in the sense that the final translation to machine code is controlled by

the explicit type and accessibility declarations that reside in the metadata of the distribution form. *Data* is also managed in the sense that objects are allocated and later deallocated by a trusted garbage collector within the runtime. The absence of explicit deallocation (and re-allocation) of memory is a precondition of any proof of type- and memory-safety in such systems. An early comparison of the two virtual machines is the paper[11].

2. Why Managed Execution?

Both *.NET* and *Java* have become key technologies in the contemporary software world. This success may appear paradoxical since both systems suffer from the same issues that doomed *ANDF*. Despite the best efforts of the "*code obfuscators*" decompilation of code is still possible. We must conclude that there are other advantages that counter the risks of including symbolic content with the distribution. The things that are different between 1989 and 2005 are the "world-wide web", web services, and the emergence of component technology.

The importance of the web to the success of managed execution platforms seems indisputable. In the case of *Java* the possibility of writing browser applets drove the early uptake of the language, while web services featured in all of the early *.NET* publicity[1]. Despite all of this emphasis on easy access to the web, and the lure of software portability, it is contended here that the real importance of managed execution derives precisely from the fact that it is *managed*.

Here is the main claim of this paper —
Managed execution provides the only resonable basis on which the promise of component technology may be realized.

2.1. Component Technology

The term *component technology* has acquired somewhat overloaded semantics so it should be clarified that in this paper the term is used in the sense of Szyperski[12]. That is — "*Software components are binary units of independent production, acquisition and deployment that interact to form a functioning system*". Components are thus —

- units of independent deployment
- units of third-party composition
- deployed in binary form

The key issue of component technology, in this context, is the software engineering means by which third parties may compose binary components to create programs that are robust and perhaps even correct. It might be added that a further challenge is to ensure that such programs continue to operate correctly in the face of the evolution of their component parts. This last element is discussed in Section 4.

The traditional means by which software complexity has been tamed is by the use of *abstraction*. That is, parts of the program are replaced by abstract representations, thus limiting the domain of analysis that is required to reason about the behaviour of the whole. Implicit in the validity of this approach is the naïve belief that the abstraction captures *all* of the interactions that propagate across the boundaries of the program parts. Many of the advances in programming languages in the last 30 years have been introduced to progressively increase the accuracy of the abstract representations and hence reduce the naïvity of the belief. Three brief examples will suffice to make the point —

- modular languages guarantee that functions may only be called with correctly typed arguments
- fully type-safe languages guarantee that pointer referents can only be of the declared type
- languages with declarative accessibilty control enforce the need to know principle

Every practising software engineer is familiar with the mayhem that results when these guarantees break down, as a result of memory deallocation faults for example.

Perhaps the most elaborate example of this approach to software design is the "design by contract" methodology incorporated into the programming language *Eiffel*. In this case software parts may be annotated with contracts in the form of preconditions, postconditions and invariants. These contracts are then enforced by a mixture of compile-time and run-time checks. The evidence seems to be that such mechanisms do indeed allow extremely robust and trustworthy software to be constructed.

All of this, so far, has been good news about which software engineers may be justly proud. The bad news is that all of the guarantees and safeguards described above are virtually useless in the context of component software!

Consider the simple example of a program component which depends for its correctness on the fact that a particular field of some object type may only be changed by the code at one program point. Such fields are safeguarded by being declared *private*. Unfortunately, if references to the enclosing object are accessible to other components the field may be mutated either through program error or by malicious intent. Declarative privacy counts for nothing in a binary component environment. Recall, for example, that the buffer overflow exploits that are a commonplace in vandalware seek to mutate function return addresses. In such cases the target location is so "private" that high level languages do not even have a mechanism to refer to the datum.

1 Indeed the last of the code-names used within Microsoft for what became *.NET* was the excruciating *NGWS*, an initialism for "next generation web services".

The issue is that all of the safeguards based on programming language mechanisms depend on the compiler for enforcement. With binary deployment of components, particularly those produced by third parties, neither the compiler nor the integrity of the deployment mechanism are a given. This is precisely the problem that managed execution uses *verification* to solve.

2.2. Safety and Security

It is important to distinguish between the separate concerns of *safety* and *security*. Memory safety and its prerequisite type-safety are necessary preconditions for forms of program analysis based on abstraction. We need to be able to reason piecewise about the behaviour of program components secure in the knowledge that components do not invalidate each others declarative invariants. This *safety* guarantee is precisely that which verified, managed execution provides.

Security is quite another matter. Both of the managed execution systems that we consider provide security mechanisms that regulate the security-relevant actions that particular components may perform. It is interesting to note that checking of security permissions requires a costly traversal of the whole chain of activation records, that is, a *stack walk*. This is necessary since it is not the permissions of a particular function that is in doubt, but the permissions of the complete chain of callers on whose behalf the function has been invoked.

Security is a separate concern to memory safety. The fact that the verifier has guaranteed the type-safety of an applet is of little consolation after the applet has reformatted the disk. Nevertheless, the enforcement of memory safety is a necessary foundation for a separate security mechanism. Consider the possible modes of attack against a stack-walking security permission checker. One exploit would be to falsify the permissions that a piece of code possesses. Another might be to falsify the call chain record by overwriting a return address. Both of these attacks are impossible in a verified, managed execution environment.

A more plausible security exploit involves managed code calling out to native (unmanaged) code. Once beyond the oversight of the verifier, anything goes. For this reason permission to invoke unmanaged code must be carefully guarded and sparingly granted in managed execution systems.

Some people find it somewhat artificial that discussions of such matters as type-safety are conducted in the language of conflict. We reason about "attackers" and try to remove "security vulnerabilities". This is neither a sign of paranoia, nor a preoccupation with fantasy games. If the invariants of a component are safe against an attacker with malicious intent, then the same invariants are safe against accidental violation by program errors in other components.

3. Some Research Issues

Managed execution systems, as they currently exist, enforce the constraints of the type declarations of the programs that they execute. This is sufficient to ensure memory safety of programs, and to ensure the absence of certain kinds of interference between components. To achieve even this is an important advance, however the kinds of invariants that can be guaranteed by such mechanisms are *syntactic* and *static*. There is a fascinating spectrum of open research possibilities that might broaden the range of program properties that managed execution might ensure.

There are also interesting research issues that have to do with the implementation of such systems.

3.1. Implementation Issues

Publicly accessible source code for both *Java* and *CLR* implementations exist, facilitating research on language compilers and *JIT* compilers. The question of which optimizations should be performed by each kind of compiler is still the subject of some experimentation, and could very well have a different answer for the *CLR* and the *JVM*.

Reliance on just-in-time compilation also brings with it the possibility, or should that be the challenge, of using the extra information available at runtime to generate faster code than is possible in an "ahead-of-time" compiler. The field of dynamic compilation and optimization is very active one with products starting to move from the laboratory to the mainstream.

A more basic kind of investigation involves the mechanisms of verification. It turns out that the algorithm specified for verification in *Java* can become computationally costly in some pathological cases. Alternative methods of verification based on "proof carrying code" seem promising.

Of course, it is always necessary to ensure correctness of the algorithms (and of their implementation) that managed execution relies on. The issues can be subtle. Here is a favourite example that nicely illustrates the subtle issues involved. One of the features of the Common Type System (*CTS*) of the *CLR* is the possibility to mark instance fields of structured types as *initonly*. The idea is that such fields are initialized at object creation, and are afterward immutable. This is a really useful feature in practice, since programs may be designed to use such fields to hold identity data, permissions and the like. In *C#* we mark such fields as *readonly* —

```
public class C {
    public readonly long serialNm;
    ...
    public C(long sn) { // Constructor
       serialNm = sn;   // assign immutable value
       ...
```

Compiling for the .NET Common Language Runtime[13] correctly warned that in the first release of *.NET* the *C#* compiler enforced this constraint but the verifier did not. As might be hoped, later releases of the *CLR* refuse verification to programs that attempt to mutate an *initonly* field.

In the description of what *initonly* means, the wording "... and is afterward immutable" seems perfectly clear, but is not the kind of rule that a verifier may check directly. What we need is an *operational* formulation of a test that checks this constraint. Here is a candidate set —

- *initonly* fields may only be set within a constructor for their enclosing type
- constructors may only be called as part of the creation of an object of the type, or as part of the creation of an object of a derived type

The "or as" part of the second rule is necessary, since when an object of a derived type is being constructed the (perhaps private) fields of the base type must be initialized by invoking a base class constructor on the newborn object of the derived type.

It turns out that the candidate rule set is insufficient, since it does not prevent a constructor from being called more than once on the same object. Here is an exploit[2] which mutates an *initonly* field, even in the presence of the candidate rule set —

```
public class D : C {      // D derives from C
  ...
  public D(C victim, long newVal) {
    ldarg.0               // push 'this' ref.
    ldarg.2               // push newVal arg
    call instance void C::'.ctor'(int64)
    ldarg.1               // push victim ref.
    ldarg.2               // push newVal arg
    call instance void C::'.ctor'(int64)
  }
```

The body of the constructor for type D is shown in Common Intermediate Language (*CIL*), where calls to constructors use the invariable name "*.ctor*".

D is a dummy class, we only use objects of this type to do our dirty work. The trick is that we have passed in the victim object of type C as an argument to the dummy constructor. The second rule above does not forbid us from passing this argument to a call of the base class constructor along with the new value for the supposedly immutable field. We cannot express this behaviour in *C#*, so the body of the above code snippet shows how it reads in *CIL* language. As a former, security-guru colleague of mine used to say "To be good at this stuff you need to have the criminal mind". In any case the third rule that is needed is —

- base class constructors may only be invoked on the newborn object within a constructor for a derived class

The first three lines of the constructor body in the code snippet are legal, and indeed are compulsory. These lines invoke the base class constructor on "arg.0", which is the location of the reference to the object under construction. The rather similar looking second call in the code is illegal according to this new rule as "arg.1" is the incoming argument, that is, the intended victim of the exploit[3].

3.2. More Expressive Type Systems

One approach to strengthening the guarantees of managed execution involves extension of the type-system. Another category of research involves the addition of such things as program assertions and protocol checks to the platform-enforced repetoire.

In essence, current managed execution systems enforce the declarative constraints of the type systems of their hosted programs. In principle any declarative aspect of a type system that is capable of being checked by an effective procedure might be added to the execution engine.

Here is a simple example. Languages such as *Ada* and *Pascal* provide for the declaration of *subrange* types, the values of which are restricted ranges of some whole-number type. It is usual for the compiler to ensure that every assignment of a new value to a datum declared to be of such a type respects the constraint. In a single language, known compiler environment consumers of such types do not need to perform range tests on values of the type. In a multi-vendor component environment no such trust could be justified, but a managed execution engine could statically guarantee enforcement of the value constraints. As it turns out neither of our example managed platforms provides for subrange types in their underlying type system, and it is hard to make a strong case for such an introduction, given the low cost of range testing at the point of use.

The more interesting issue of execution engine enforcement of program invariants such as pre- and post-conditions has received some attention. Nam Tran's doctoral research at Monash University has involved implementing *Eiffel*-like contracts with the support of a modified version of the "shared source" version of the *CLR*.

Almost all of the enforcement of managed execution systems have to do with *static* features of the type system. Accessibility constraints, conformance to the rules of sub-type

2 I am happy to share this exploit, since it does not work any more!

3 And of course if you are wondering, overwriting "arg.0" by "arg.1" doesn't get past the verifier either!

polymorphism, and fulfilment of contracts to implement named interfaces, are the kinds of things that are guaranteed. There are a whole range of dynamic issues that relate to correctness of component systems. These dynamic rules may be expressed in terms of *protocols*. Protocols specify such things as rules that certain methods may only be called if other calls have preceeded the call in certain allowed patterns. It is known that in some cases the rules cannot be described in terms of finite state machines. The understanding of such rules is an active area of research. An associated open question is — how can such protocols be enforced within a component framework that allows for third party composition of systems?

Finally, it may be noted that both of our example managed execution systems have announced enhancements to their underlying type systems to support *parametric polymorphism*, or "generics" as it is more commonly called. Sun Microsystems and Microsoft have adopted very different implementation strategies for this very significant enhancement. Sun has chosen to take a less efficient implementation mechanism, but one that leaves the *JVM* unchanged. Microsoft has enhanced the *CLR* with some additional instructions and lots of new metadata so that the *JIT* can specialize code for particular instantiations of generic types.

4. Version Evolution

Software evolution is one of the difficult issues of our time. As many an elderly *COBOL* programmer remarked in the late 1990s "Nobody expected this software to be around for so long". In the past the problems have been less for monolithic software, particularly where programs are statically linked. However for distributed software, and even more so for component software the problems of version evolution have become acute. This is a problem that has received a lot of attention within the Microsoft world, but no component system can ignore these issues.

The rest of this section summarizes some informal discussions. Credit for the key ideas of these proposals belongs to Chris Brumme, Patrick Dussud, Anders Hejlsberg, Jim Miller, Clemens Szyperski, Tony Williams and others within Microsoft. Raising of these topics publicly should not be taken as any kind of endorsement of the proposal by Microsoft, nor as a committment to implement any of these ideas in any future product.

4.1. The Perils of Registry

One of the most difficult problems for component based systems is that of version evolution. The problem is familiar to *Windows* system administrators. Pre-*.NET* component systems share dynamically linked libraries (*DLLs*) the identity and location of which is held in a global registry. A typical problem arises when a newly installed application brings with it a new version of a *DLL* that is used by an existing application. After installation of the new program some apparently unrelated program breaks. Re-installation of the broken program restores that program's functionality, but the previously installed program now does not work. This situation is colloquially known as "*DLL Hell*". It is caused by a failure of backward compatibility in the evolution of the library that is shared by the two applications.

Such a backward compatability failure does not necessarily indicate incompetence on the part of the software provider. It is an unfortunate fact that programs sometimes depend on library behaviours that are outside that specified in the application programming interface (*API*), that is, they rely on undocumented behaviour. Furthermore it is sometimes necessary to modify even the documented behaviour, for example to eliminate a security vulnerability. In any case the problem is particularly difficult in systems that rely on global registries.

The problems of *DLL-hell* are lessened in the *.NET* framework, which provides for "side-by-side" execution. In this system the identity of loadable assemblies depends on a four-part version number, and a cryptographically strong originator signature. Every application may set a policy that allows it to choose between the latest version of a shared library, or to insist on one exact version. There are several intermediate policy possibilities. All of the various library versions may co-exist in the "global assembly cache" (*GAC*), and simultaneously executing applications may run different versions of the same library "side-by-side" as the name implies. More to the point for component based systems different components of the same application may use different versions of the same library. This possibility effectively uncouples the version dependencies of the different components.

The *.NET* system makes *DLL-hell* a thing of the past, or at least a thing of a rapidly receding present. Sadly however any belief that the version evolution problem is now fully solved is premature. The problem is that not *all* assemblies may be executed side-by-side. For example, if a library controls some unique resource of the machine then only one version may run concurrently. Such a library *must* be shared, and different versions cannot execute side-by-side. Worse still, as more of the operating system software migrates to managed code, more of the system-supplied objects will lock in particular versions of their defining types.

Conflicts between components may be indirect. Suppose two components depend on different versions of the same shared library, A say. Let us further suppose that library A is intended to to permit side-by-side execution. Unfortunately, if different versions of A depend on different versions of some second library, B say, then if B does not support side-by-side execution, then neither can A. It seems

that as *DLL-hell* recedes, *GAC-hell* looms on the horizon.

4.2. Platform and Library Types

It is likely that the problems of version evolution are fundamentally intractable, but there are some interesting approaches for at least *managing* the problem. In particular, it is important to lessen the *domino effect* caused by chains of dependencies between library components as described above.

One possibility receiving some debate currently involves adding a new declarative attribute to type definitions. At its simplest the idea is to mark every type as being either a *platform* type or a *library* type. Each denotation implies a contractual obligation as to future evolution. Library types are free to evolve between versions, but guarantee that different versions will be able to execute side-by-side. Platform types are bound to a much higher level of compatability. Applications do not have a choice as to the version of a platform type that they use. As the name implies, the software must use whatever version of the type that the platform supplies, and all components on the machine will use the same version.

Platform types are not capable of side-by-side execution, either because they depend on a non-sharable resource, or because they are locked by a dependency on the underlying operating system or even the *CLR* version. For example the character string type *System.String* must be a platform type, since the type is built in to the execution engine.

Dependencies The constraints on the dependencies between the two categories of types can be easily deduced.

Library types may freely depend on platform types. They may use platform types in the implementation of their own behaviour, and may expose platform types in their visible interface. Any such dependency does not constrain the evolution of the type, nor does it add further dependencies to users of the type. Library types may also depend on other library types. This will make the type dependent on a particular version of the other types. If the "other" library types are not exposed in the visible interface of the type, then the *users* of the type do not become contaminated by spurious version dependencies. In a typical scenario several library types would be defined in a single library. These types will depend on each other, but will evolve together, compatibly, as their containing library evolves.

Platform types, on the other hand, are bound by a stricter regime. A platform type may depend on library types in its implementation, if necessary it can deploy with the library version that it requires. However, a platform type must never expose any library type in its visible interface. Thus for such types every public field, every formal argument of a public method and every return type must be a platform type. A platform type cannot derive from a base class that is a library type, nor may it implement an interface that is a library type. Rather less obviously platform types may not allow library methods to escape in (for example) arrays of *System.Object* or onto a system clipboard.

Using Platform and Library Types The separation between platform and library types only becomes significant at the boundaries of software components. Creators of components will expect their components to have to use whatever version of the platform types that the platform offers. But since every other component of the application will necessarily use the same version, there is no possibility of conflict.

Components may use library types of their own choosing to implement their own behaviour. They will thereby acquire a dependency on a particular library or libraries, but they can deploy with the version of the library that they depend on, and lock down that exact version if necessary. Provided that the component does not expose the dependency to its users, there will be no conflict if another component has locked in a different version of the same library type. In effect, these constraints mean that components must interact and communicate using only platform types. If, contrary to this advice, components communicate by exchanging library types then the components must agree to use the same exact library version. This clearly places a very strong limitation on third-party composition of such components.

Designing Platform Types Platform types appear to be more versatile, so it may appear attractive to make as many types as possible platform types. This is not a good idea. The evolution of platform types is necessarily slow and painstaking. Platform types have contracted to maintain an almost impossibly high level of backward compatability. When a new version is released every existing application will have to use it ... so it better just work. We may conclude that an extremely high level of quality assurance will be needed to maintain platform types, and this will be expensive.

The challenge is that the required compatibility for platform types is not just at the level of using the same method signatures in the binary form, but at the *behavioural* level. Every aspect of documented behaviour must be maintained in new versions of the type. Conversely, *any* aspect of undocumented behaviour visible to users provides the opportunity for user code to be broken by future version evolution.

Choosing to create a platform type should thus be approached with some caution. Since the users of the code will be very upset if the behaviour ever changes the design has to be right first time. And then, having designed it right, the company needs to expensively maintain the type for as long as it plans to stay in business.

4.3. But Will it Fly?

The separation of types into categories according to the style of their evolution may or may not find its way into the type systems of the mainstream managed platforms. Nevertheless, the very idea of such a separation is an important tool in tackling versioning issues in component systems. The concept, and the design rules that flow from it, have a wider applicability to all large scale systems for which it is necessary to upgrade subsystems piecewise. In that case, even if all the sub-systems come from a single vendor, new subsystem versions must interwork with other subsystems that will be replaced at later times. The key lesson is to tightly control the evolution of the types that cross the subsystem boundaries.

Finally it may be noted that within the component world, with third-party composition of independently developed components, the type-safety guarantees of managed execution are critical to controlling the versioning problem. It is also relevant to observe that managed execution systems restrict the visibility of implementation artifacts, and thus make it less likely that the user of a platform type can depend on undocumented features of a particular version of the type.

5. Concluding Remarks

This quick overview has reviewed the historical context in which abstract machines have morphed into the currently popular managed execution systems. The recital of the history goes some way towards explaining how the current systems came to use abstract machines to represent program behaviour. Nevertheless, the notion of managed execution (or alternatively *metadata mediated execution*) and the use of abstract stack machine representations are independent. In fact it may be argued that some slight advantage might be gained by using a program representation other than the instructions for an abstract stack machine. Even the advantage of high program density traditionally claimed for stack machines seems dubious, given the significant volume of metadata that must accompany the bytecodes in these systems.

Irrespective of the choice of instruction set, managed execution systems with their reliance on symbolic metadata, provide representations of program behaviour that are suitably abstracted from the details of any particular machine. In principle they provide for a level of software portability that goes beyond anything previously achieved. This is a one reason for the importance of such systems, but not the most important.

The real importance of managed execution systems, and their critical role in the future of software development depends on the fact that they are *managed*. Managed execution provides for the enforcement of type- and memory-safety in environments where the integrity of neither the originating compiler nor the deployment mechanism may be guaranteed. Memory safety is, in turn, the guarantee that is required to ensure lack of interference between program parts in a component framework. The enforcement of type-system invariants is also an essential factor in managing version evolution.

References

[1] K. V. Nori, U. Ammann, K. Jensen, H. H. Nageli and Ch. Jacobi, "Pascal-P Implementation Notes" Ch. 9 in D. W. Barron (Ed), *Pascal – the Language and its Implementation*. J Wiley, 1981.

[2] K. Bowles, *Beginner's Guide for the UCSD Pascal System*, Byte Books, 1980.

[3] D. R. Perkins and R. L. Sites, "Machine-indpendent Pascal code optimization". Proc. 1979 SIGPLAN symposium on Compiler Construction, ACM, 1979.

[4] K. John Gough, "Multi-language, Multi-target Compiler Development", JMLC, Linz Austria, March 1997. Also in *Modular Programming Languages*, H. Mössenböck (Ed), LNCS No. 1204, Springer Verlag.

[5] Hassan Aït-Kaci, *Warren's Abstract Machine: A Tutorial Reconstruction*. MIT Press, 1991.

[6] S. Macrakis, "From UNCOL to ANDF: Progress in Standard Intermediate Languages." Technical Report, Open Software Foundation Research Institute, 1993.

[7] M. Franz, *Code Generation On-the-Fly: A Key to Portable Software*. Doctoral Dissertation No. 10497, ETH Zurich, March 1994.

[8] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*. Addison-Wesley, Reading MA, 1997.

[9] Sun Microsystems, "Java Technology: the Early Years" http://java.sun.com/features/1998/05/birthday.html

[10] J. Miller and S. Ragsdale, *The Common Language Infrastructure Annotated Standard*. Addison-Wesley, New York, NY, 2004.

[11] K. John Gough, "Stacking them up: a Comparison of Virtual Machines". Australian Computer Systems and Architecture Conference (ACSAC-2001), Gold Coast, Australia, February 2001.

[12] C. Szyperski, *Component Software: Beyond Object Oriented Programming*. ACM Press and Addison-Wesley, New York, NY, 1998.

[13] J. Gough, *Compiling for the .NET Common Language Runtime*. Prentice-Hall, Saddle River, NJ, 2002.

EXHIBIT H

A Brief History of Just-In-Time

JOHN AYCOCK

University of Calgary

Software systems have been using "just-in-time" compilation (JIT) techniques since the 1960s. Broadly, JIT compilation includes any translation performed dynamically, after a program has started execution. We examine the motivation behind JIT compilation and constraints imposed on JIT compilation systems, and present a classification scheme for such systems. This classification emerges as we survey forty years of JIT work, from 1960–2000.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors; K.2 [**History of Computing**]: *Software*

General Terms: Languages, Performance

Additional Key Words and Phrases: Just-in-time compilation, dynamic compilation

1. INTRODUCTION

> Those who cannot remember the past are condemned to repeat it.
>
> *George Santayana, 1863–1952* [Bartlett 1992]

This oft-quoted line is all too applicable in computer science. Ideas are generated, explored, set aside—only to be reinvented years later. Such is the case with what is now called "just-in-time" (JIT) or dynamic compilation, which refers to translation that occurs after a program begins execution.

Strictly speaking, JIT compilation systems ("JIT systems" for short) are completely unnecessary. They are only a means to improve the time and space efficiency of programs. After all, the central problem JIT systems address is a solved one: translating programming languages into a form that is executable on a target platform.

What is translated? The scope and nature of programming languages that require translation into executable form covers a wide spectrum. Traditional programming languages like Ada, C, and Java are included, as well as little languages [Bentley 1988] such as regular expressions.

Traditionally, there are two approaches to translation: compilation and interpretation. Compilation translates one language into another—C to assembly language, for example—with the implication that the translated form will be more amenable to later execution, possibly after further compilation stages. Interpretation eliminates these intermediate steps, performing the same analyses as compilation, but performing execution immediately.

This work was supported in part by a grant from the National Science and Engineering Research Council of Canada.

Author's address: Department of Computer Science, University of Calgary, 2500 University Dr. N. W., Calgary, Alta., Canada T2N 1N4; email: aycock@cpsc.ucalgary.ca.

JIT compilation is used to gain the benefits of both (static) compilation and interpretation. These benefits will be brought out in later sections, so we only summarize them here:

—Compiled programs run faster, especially if they are compiled into a form that is directly executable on the underlying hardware. Static compilation can also devote an arbitrary amount of time to program analysis and optimization. This brings us to the primary constraint on JIT systems: speed. A JIT system must not cause untoward pauses in normal program execution as a result of its operation.

—Interpreted programs are typically smaller, if only because the representation chosen is at a higher level than machine code, and can carry much more semantic information implicitly.

—Interpreted programs tend to be more portable. Assuming a machine-independent representation, such as high-level source code or virtual machine code, only the interpreter need be supplied to run the program on a different machine. (Of course, the program still may be doing nonportable operations, but that's a different matter.)

—Interpreters have access to run-time information, such as input parameters, control flow, and target machine specifics. This information may change from run to run or be unobtainable prior to run-time. Additionally, gathering some types of information about a program before it runs may involve algorithms which are undecidable using static analysis.

To narrow our focus somewhat, we only examine software-based JIT systems that have a nontrivial translation aspect. Keppel et al. [1991] eloquently built an argument for the more general case of run-time code generation, where this latter restriction is removed.

Note that we use the term *execution* in a broad sense—we call a program representation executable if it can be executed by the JIT system in any manner, either directly as in machine code, or indirectly using an interpreter.

2. JIT COMPILATION TECHNIQUES

Work on JIT compilation techniques often focuses around implementation of a particular programming language. We have followed this same division in this section, ordering from earliest to latest where possible.

2.1. Genesis

Self-modifying code has existed since the earliest days of computing, but we exclude that from consideration because there is typically no compilation or translation aspect involved.

Instead, we suspect that the earliest published work on JIT compilation was McCarthy's [1960] LISP paper. He mentioned compilation of functions into machine language, a process fast enough that the compiler's output needn't be saved. This can be seen as an inevitable result of having programs and data share the same notation [McCarthy 1981].

Another early published reference to JIT compilation dates back to 1966. The University of Michigan Executive System for the IBM 7090 explicitly notes that the assembler [University of Michigan 1966b, p. 1] and loader [University of Michigan 1966a, p. 6] can be used to translate and load during execution. (The manual's preface says that most sections were written before August 1965, so this likely dates back further.)

Thompson's [1968] paper, published in *Communications of the ACM*, is frequently cited as "early work" in modern publications. He compiled regular expressions into IBM 7094 code in an ad hoc fashion, code which was then executed to perform matching.

2.2. LC^2

The Language for Conversational Computing, or LC^2, was designed for interactive programming [Mitchell et al. 1968]. Although used briefly at Carnegie-Mellon University for teaching, LC^2 was

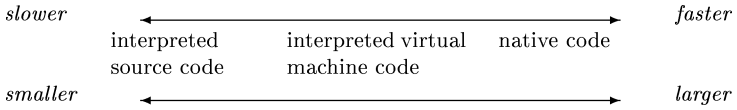

Fig. 1. The time-space tradeoff.

primarily an experimental language [Mitchell 2000]. It might otherwise be consigned to the dustbin of history, if not for the techniques used by Mitchell in its implementation [Mitchell 1970], techniques that later influenced JIT systems for Smalltalk and Self.

Mitchell observed that compiled code can be derived from an interpreter at run-time, simply by storing the actions performed during interpretation. This only works for code that has been executed, however—he gave the example of an if-then-else statement, where only the else-part is executed. To handle such cases, code is generated for the unexecuted part which reinvokes the interpreter should it ever be executed (the then-part, in the example above).

2.3. APL

The seminal work on efficient APL implementation is Abrams' dissertation [Abrams 1970]. Abrams concocted two key APL optimization strategies, which he described using the connotative terms *drag-along* and *beating*. Drag-along defers expression evaluation as long as possible, gathering context information in the hopes that a more efficient evaluation method might become apparent; this might now be called *lazy evaluation*. Beating is the transformation of code to reduce the amount of data manipulation involved during expression evaluation.

Drag-along and beating relate to JIT compilation because APL is a very dynamic language; types and attributes of data objects are not, in general, known until run-time. To fully realize these optimizations' potential, their application must be delayed until run-time information is available.

Abrams' "APL Machine" employed two separate JIT compilers. The first translated APL programs into postfix code for a D-machine,[1] which maintained a buffer of deferred instructions. The D-machine acted as an "algebraically simplifying compiler" [Abrams 1970, p. 84] which would perform drag-along and beating at run-time, invoking an E-machine to execute the buffered instructions when necessary.

Abrams' work was directed toward an architecture for efficient support of APL, hardware support for high-level languages being a popular pursuit of the time. Abrams never built the machine, however; an implementation was attempted a few years later [Schroeder and Vaughn 1973].[2] The techniques were later expanded upon by others [Miller 1977], although the basic JIT nature never changed, and were used for the software implementation of Hewlett-Packard's APL\3000 [Johnston 1977; van Dyke 1977].

2.4. Mixed Code, Throw-Away Code, and BASIC

The tradeoff between execution time and space often underlies the argument for JIT compilation. This tradeoff is summarized in Figure 1. The other consideration is that most programs spend the majority of time executing a minority of code, based on data from empirical studies [Knuth 1971]. Two ways to reconcile these observations have appeared: mixed code and throw-away compiling.

Mixed code refers to the implementation of a program as a mixture of native code and interpreted code, proposed independently by Dakin and Poole [1973] and Dawson [1973]. The frequently executed parts of the program would be

[1] Presumably *D* stood for *Deferral* or *Drag-Along*.

[2] In the end, Litton Industries (Schroeder and Vaughn's employer) never built the machine [Mauriello 2000].

in native code, the infrequently executed parts interpreted, hopefully yielding a smaller memory footprint with little or no impact on speed. A fine-grained mixture is implied: implementing the program with interpreted code and the libraries with native code would *not* constitute mixed code.

A further twist to the mixed code approach involved customizing the interpreter [Pittman 1987]. Instead of mixing native code into the program, the native code manifests itself as special virtual machine instructions; the program is then compiled entirely into virtual machine code.

The basic idea of mixed code, switching between different types of executable code, is still applicable to JIT systems, although few researchers at the time advocated generating the machine code at run-time. Keeping both a compiler and an interpreter in memory at run-time may have been considered too costly on the machines of the day, negating any program size tradeoff.

The case against mixed code comes from software engineering [Brown 1976]. Even assuming that the majority of code will be shared between the interpreter and compiler, there are still two disparate pieces of code (the interpreter proper and the compiler's code generator) which must be maintained and exhibit identical behavior.

(Proponents of partial evaluation, or program specialization, will note that this is a specious argument in some sense, because a compiler can be thought of as a specialized interpreter [Jones et al. 1993]. However, the use of partial evaluation techniques is not currently widespread.)

This brings us to the second manner of reconciliation: throw-away compiling [Brown 1976]. This was presented purely as a space optimization: instead of static compilation, parts of a program could be compiled dynamically on an as-needed basis. Upon exhausting memory, some or all of the compiled code could be thrown away; the code would be regenerated later if necessary.

BASIC was the testbed for throw-away compilation. Brown [1976] essentially characterized the technique as a good way to address the time-space tradeoff; Hammond [1977] was somewhat more adamant, claiming throw-away compilation to be superior except when memory is tight.

A good discussion of mixed code and throw-away compiling may be found in Brown [1990].

2.5. FORTRAN

Some of the first work on JIT systems where programs automatically optimize their "hot spots" at run-time was due to Hansen [1974].[3] He addressed three important questions:

(1) What code should be optimized? Hansen chose a simple, low-cost frequency model, maintaining a frequency-of-execution counter for each block of code (we use the generic term *block* to describe a unit of code; the exact nature of a block is immaterial for our purposes).

(2) When should the code be optimized? The frequency counters served a second rôle: crossing a threshold value made the associated block of code a candidate for the next "level" of optimization, as described below. "Supervisor" code was invoked between blocks, which would assess the counters, perform optimization if necessary, and transfer control to the next block of code. The latter operation could be a direct call, or interpreter invocation—mixed code was supported by Hansen's design.

(3) How should the code be optimized? A set of conventional machine-independent and machine-dependent optimizations were chosen and ordered, so a block might first be optimized by constant folding, by common subexpression elimination the second

[3] Dawson [1973] mentioned a 1967 report by Barbieri and Morrissey where a program begins execution in interpreted form, and frequently executed parts "can be converted to machine code." However, it is not clear if the conversion to machine code occurred at run-time. Unfortunately, we have not been able to obtain the cited work as of this writing.

time optimization occurs, by code motion the third time, and so on. Hansen [1974] observed that this scheme limits the amount of time taken at any given optimization point (especially important if the frequency model proves to be incorrect), as well as allowing optimizations to be incrementally added to the compiler.

Programs using the resulting Adaptive FORTRAN system reportedly were not always faster than their statically compiled-and-optimized counterparts, but performed better overall.

Returning again to mixed code, Ng and Cantoni [1976] implemented a variant of FORTRAN using this technique. Their system could compile functions at run-time into "pseudo-instructions," probably a tokenized form of the source code rather than a lower-level virtual machine code. The pseudo-instructions would then be interpreted. They claimed that run-time compilation was useful for some applications and avoided a slow compile-link process. They did not produce mixed code at run-time; their use of the term referred to the ability to have statically compiled FORTRAN programs call their pseudo-instruction interpreter automatically when needed via linker trickery.

2.6. Smalltalk

Smalltalk source code is compiled into virtual machine code when new methods are added to a class [Goldberg and Robson 1985]. The performance of naïve Smalltalk implementations left something to be desired, however.

Rather than attack the performance problem with hardware, Deutsch and Schiffman [1984] made key optimizations in software. The observation behind this was that they could pick the most efficient representation for information, so long as conversion between representations happened automatically and transparently to the user.

JIT conversion of virtual machine code to native code was one of the optimization techniques they used, a process they likened to macro-expansion. Procedures were compiled to native code lazily, when execution entered the procedure; the native code was cached for later use. Their system was linked to memory management in that native code would never be paged out, just thrown away and regenerated later if necessary.

In turn, Deutsch and Schiffman [1984] credited the dynamic translation idea to Rau [1978]. Rau was concerned with "universal host machines" which would execute a variety of high-level languages well (compared to, say, a specialized APL machine). He proposed dynamic translation to microcode at the granularity of single virtual machine instructions. A hardware cache, the dynamic translation buffer, would store completed translations; a cache miss would signify a missing translation, and fault to a dynamic translation routine.

2.7. Self

The Self programming language [Ungar and Smith 1987; Smith and Ungar 1995], in contrast to many of the other languages mentioned in this section, is primarily a research vehicle. Self is in many ways influenced by Smalltalk, in that both are pure object-oriented languages—everything is an object. But Self eschews classes in favor of prototypes, and otherwise attempts to unify a number of concepts. Every action is dynamic and changeable, and even basic operations, like local variable access, require invocation of a method. To further complicate matters, Self is a dynamically-typed language, meaning that the types of identifiers are not known until run-time.

Self's unusual design makes efficient implementation difficult. This resulted in the development of the most aggressive, ambitious JIT compilation and optimization up to that time. The Self group noted three distinct generations of compiler [Hölzle 1994], an organization we follow below; in all cases, the compiler was invoked dynamically upon a method's invocation, as in Deutsch and Schiffman's [1984] Smalltalk system.

2.7.1. First Generation. Almost all the optimization techniques employed by Self compilers dealt with type information, and transforming a program in such a way that some certainty could be had about the types of identifiers. Only a few techniques had a direct relationship with JIT compilation, however.

Chief among these, in the first-generation Self compiler, was customization [Chambers et al. 1989; Chambers and Ungar 1989; Chambers 1992]. Instead of dynamically compiling a method into native code that would work for any invocation of the method, the compiler produced a version of the method that was customized to that particular context. Much more type information was available to the JIT compiler compared to static compilation, and by exploiting this fact the resulting code was much more efficient. While method calls from similar contexts could share customized code, "overcustomization" could still consume a lot of memory at run-time; ways to combat this problem were later studied [Dieckmann and Hölzle 1997].

2.7.2. Second Generation. The second-generation Self compiler extended one of the program transformation techniques used by its predecessor, and computed much better type information for loops [Chambers and Ungar 1990; Chambers 1992].

This Self compiler's output was indeed faster than that of the first generation, but it came at a price. The compiler ran 15 to 35 times more slowly on benchmarks [Chambers and Ungar 1990, 1991], to the point where many users refused to use the new compiler [Hölzle 1994]!

Modifications were made to the responsible algorithms to speed up compilation [Chambers and Ungar 1991]. One such modification was called *deferred compilation of uncommon cases*.[4] The compiler is informed that certain events, such as arithmetic overflow, are unlikely to occur. That being the case, no code is generated for these uncommon cases; a stub is left in the code instead, which will invoke the compiler again if necessary. The practical result of this is that the code for uncommon cases need not be analyzed upon initial compilation, saving a substantial amount of time.[5]

Ungar et al. [1992] gave a good presentation of optimization techniques used in Self and the resulting performance in the first- and second-generation compilers.

2.7.3. Third Generation. The third-generation Self compiler attacked the issue of slow compilation at a much more fundamental level. The Self compiler was part of an interactive, graphical programming environment; executing the compiler on-the-fly resulted in a noticeable pause in execution. Hölzle argued that measuring pauses in execution for JIT compilation by timing the amount of time the compiler took to run was deceptive, and not representative of the user's experience [Hölzle 1994; Hölzle and Ungar 1994b]. Two invocations of the compiler could be separated by a brief spurt of program execution, but would be perceived as one long pause by the user. Hölzle compensated by considering temporally related groups of pauses, or "pause clusters," rather than individual compilation pauses.

As for the compiler itself, compilation time was reduced—or at least spread out—by using adaptive optimization, similar to Hansen's [1974] FORTRAN work. Initial method compilation was performed by a fast, nonoptimizing compiler; frequency-of-invocation counters were kept for each method to determine when recompilation should occur [Hölzle 1994; Hölzle and Ungar 1994a, 1994b]. Hölzle makes an interesting comment on this mechanism:

> . . . in the course of our experiments we discovered that the trigger mechanism ("when") is

[4] In Chambers' thesis, this is referred to as "lazy compilation of uncommon branches," an idea he attributes to a suggestion by John Maloney in 1989 [Chambers 1992, p. 123]. However, this is the same technique used in Mitchell [1970], albeit for different reasons.

[5] This technique can be applied to dynamic compilation of exception handling code [Lee et al. 2000].

> much less important for good recompilation results than the selection mechanism ("what"). [Hölzle 1994, p. 38][6]

This may come from the slightly counterintuitive notion that the best candidate for recompilation is *not* necessarily the method whose counter triggered the recompilation. Object-oriented programming style tends to encourage short methods; a better choice may be to (re)optimize the method's caller and incorporate the frequently invoked method inline [Hölzle and Ungar 1994b].

Adaptive optimization adds the complication that a modified method may already be executing, and have information (such as an activation record on the stack) that depends on the previous version of the modified method [Hölzle 1994]; this must be taken into consideration.[7]

The Self compiler's JIT optimization was assisted by the introduction of "type feedback" [Hölzle 1994; Hölzle and Ungar 1994a]. As a program executed, type information was gathered by the run-time system, a straightforward process. This type information would then be available if and when recompilation occurred, permitting more aggressive optimization. Information gleaned using type feedback was later shown to be comparable with, and perhaps complementary to, information from static type inference [Agesen and Hölzle 1995; Agesen 1996].

2.8. Slim Binaries and Oberon

One problem with software distribution and maintenance is the heterogeneous computing environment in which software runs: different computer architectures require different binary executables. Even within a single line of backward-compatible processors, many variations in capability can exist; a program statically compiled for the least-common denominator of processor may not take full advantage of the processor on which it eventually executes.

In his doctoral work, Franz addressed these problems using "slim binaries" [Franz 1994; Franz and Kistler 1997]. A slim binary contains a high-level, machine-independent representation[8] of a program module. When a module is loaded, executable code is generated for it on-the-fly, which can presumably tailor itself to the run-time environment. Franz, and later Kistler, claimed that generating code for an entire module at once was often superior to the method-at-a-time strategy used by Smalltalk and Self, in terms of the resulting code performance [Franz 1994; Kistler 1999].

Fast code generation was critical to the slim binary approach. Data structures were delicately arranged to facilitate this; generated code that could be reused was noted and copied if needed later, rather than being regenerated [Franz 1994].

Franz implemented slim binaries for the Oberon system, which allows dynamic loading of modules [Wirth and Gutknecht 1989]. Loading and generating code for a slim binary was not faster than loading a traditional binary [Franz 1994; Franz and Kistler 1997], but Franz argued that this would eventually be the case as the speed discrepancy between processors and input/output (I/O) devices increased [Franz 1994].

Using slim binaries as a starting point, Kistler's [1999] work investigated "continuous" run-time optimization, where parts of an executing program can be optimized *ad infinitum*. He contrasted this to the adaptive optimization used in Self, where optimization of methods would eventually cease.

Of course, reoptimization is only useful if a new, better, solution can be obtained; this implies that continuous optimization is best suited to optimizations whose input varies over time with the program's

[6] The same comment, with slightly different wording, also appears in Hölzle and Ungar [1994a, p. 328].

[7] Hansen's work in 1974 could ignore this possibility; the FORTRAN of the time did not allow recursion, and so activation records and a stack were unnecessary [Sebesta 1999].

[8] This representation is an abstract syntax tree, to be precise.

execution.[9] Accordingly, Kistler looked at cache optimizations—rearranging fields in a structure dynamically to optimize a program's data-access patterns [Kistler 1999; Kistler and Franz 1999]—and a dynamic version of trace scheduling, which optimizes based on information about a program's control flow during execution [Kistler 1999].

The continuous optimizer itself executes in the background, as a separate low-priority thread which executes only during a program's idle time [Kistler 1997, 1999]. Kistler used a more sophisticated metric than straightforward counters to determine when to optimize, and observed that deciding *what* to optimize is highly optimization-specific [Kistler 1999].

An idea similar to continuous optimization has been implemented for Scheme. Burger [1997] dynamically reordered code blocks using profile information, to improve code locality and hardware branch prediction. His scheme relied on the (copying) garbage collector to locate pointers to old versions of a function, and update them to point to the newer version. This dynamic recompilation process could be repeated any number of times [Burger 1997, page 70].

2.9. Templates, ML, and C

ML and C make strange bedfellows, but the same approach has been taken to dynamic compilation in both. This approach is called *staged compilation*, where compilation of a single program is divided into two stages: static and dynamic compilation. Prior to run-time, a static compiler compiles "templates," essentially building blocks which are pieced together at run-time by the dynamic compiler, which may also place run-time values into holes left in the templates. Typically these templates are specified by user annotations, although some work has been done on deriving them automatically [Mock et al. 1999].

As just described, template-based systems arguably do not fit our description of JIT compilers, since there would appear to be no nontrivial translation aspect. However, templates may be encoded in a form which requires run-time translation before execution, or the dynamic compiler may perform run-time optimizations after connecting the templates.

Templates have been applied to (subsets of) ML [Leone and Lee 1994; Lee and Leone 1996; Wickline et al. 1998]. They have also been used for run-time specialization of C [Consel and Noël 1996; Marlet et al. 1999], as well as dynamic extensions of C [Auslander et al. 1996; Engler et al. 1996; Poletto et al. 1997]. One system, Dynamo,[10] proposed to perform staged compilation and dynamic optimization for Scheme and Java, as well as for ML [Leone and Dybvig 1997].

Templates aside, ML may be dynamically compiled anyway. In Cardelli's description of his ML compiler, he noted:

> [Compilation] is repeated for every definition or expression typed by the user. . . or fetched from an external file. Because of the interactive use of the compiler, the compilation of small phrases must be virtually instantaneous. [Cardelli 1984, p. 209]

2.10. Erlang

Erlang is a functional language, designed for use in large, soft real-time systems such as telecommunications equipment [Armstrong 1997]. Johansson et al. [2000] described the implementation of a JIT compiler for Erlang, HiPE, designed to address performance problems.

As a recently designed system without historical baggage, HiPE stands out in that the user must explicitly invoke the JIT compiler. The rationale for this is that it gives the user a fine degree of control over the performance/code space tradeoff that mixed code offers [Johansson et al. 2000].

HiPE exercises considerable care when performing "mode-switches" back and

[9] Although, making the general case for run-time optimization, he discussed intermodule optimizations where this is not the case [Kistler 1997].

[10] A name collision: Leone and Dybvig's "Dynamo" is different from the "Dynamo" of Bala et al. [1999].

forth between native and interpreted code. Mode-switches may be needed at the obvious locations—calls and returns—as well as for thrown exceptions. Their calls use the mode of the *caller* rather than the mode of the called code; this is in contrast to techniques used for mixed code in Lisp (Gabriel and Masinter [1985] discussed mixed code calls in Lisp and their performance implications).

2.11. Specialization and O'Caml

O'Caml is another functional language, and can be considered a dialect of ML [Rémy et al. 1999]. The O'Caml interpreter has been the focus of run-time specialization work.

Piumarta and Riccardi [1998] specialized the interpreter's instructions to the program being run, in a limited way.[11] They first dynamically translated interpreted bytecodes into direct threaded code [Bell 1973], then dynamically combined blocks of instructions together into new "macro opcodes," modifying the code to use the new instructions. This reduced the overhead of instruction dispatch, and yielded opportunities for optimization in macro opcodes which would not have been possible if the instructions had been separate (although they did not perform such optimizations). As presented, their technique did not take dynamic execution paths into account, and they noted that it is best suited to low-level instruction sets, where dispatch time is a relatively large factor in performance.

A more general approach to run-time specialization was taken by Thibault et al. [2000]. They applied their program specializer, Tempo [Consel et al. 1998], to the Java virtual machine and the O'Caml interpreter at run-time. They noted:

> While the speedup obtained by specialization is significant, it does not compete with results obtained with hand-written off-line or run-time compilers. [Thibault et al. 2000, p. 170]

But later in the paper they stated that

> . . . program specialization is entering relative maturity. [Thibault et al. 2000, p. 175]

This may be taken to imply that, at least for the time being, program specialization may not be as fruitful as other approaches to dynamic compilation and optimization.

2.12. Prolog

Prolog systems dynamically compile, too, although the execution model of Prolog necessitates use of specialized techniques. Van Roy [1994] gave an outstanding, detailed survey of the area. One of SICStus Prolog's native code compilers, which could be invoked and have its output loaded dynamically, was described in Haygood [1994].

2.13. Simulation, Binary Translation, and Machine Code

Simulation is the process of running native executable machine code for one architecture on another architecture.[12] How does this relate to JIT compilation? One of the techniques for simulation is binary translation; in particular, we focus on dynamic binary translation that involves translating from one machine code to another at run-time. Typically, binary translators are highly specialized with respect to source and target; research on retargetable and "resourceable" binary translators is still in its infancy [Ung and Cifuentes 2000]. Altman et al. [2000b] have a good discussion of the challenges involved in binary translation, and Cmelik and Keppel [1994] compared pre-1995 simulation systems in detail. Rather than duplicating their work, we will take a higher-level view.

May [1987] proposed that simulators could be categorized by their implementation technique into three generations. To

[11] Thibault et al. [2000] provided an alternative view on Piumarta and Riccardi's work with respect to specialization.

[12] We use the term *simulate* in preference to *emulate* as the latter has the connotation that hardware is heavily involved in the process. However, some literature uses the words interchangeably.

this, we add a fourth generation to characterize more recent work.

(1) First-generation simulators were interpreters, which would simply interpret each source instruction as needed. As might be expected, these tended to exhibit poor performance due to interpretation overhead.

(2) Second-generation simulators dynamically translated source instructions into target instruction one at a time, caching the translations for later use.

(3) Third-generation simulators improved upon the performance of second-generation simulators by dynamically translating entire blocks of source instructions at a time. This introduces new questions as to what should be translated. Most such systems translated either basic blocks of code or extended basic blocks [Cmelik and Keppel 1994], reflecting the static control flow of the source program. Other static translation units are possible: one anomalous system, DAISY, performed page-at-a-time translations from PowerPC to VLIW instructions [Ebcioğlu and Altman 1996, 1997].

(4) What we call fourth-generation simulators expand upon the third-generation by dynamically translating paths, or traces. A path reflects the control flow exhibited by the source program at run-time, a dynamic instead of a static unit of translation. The most recent work on binary translation is concentrated on this type of system.

Fourth-generation simulators are predominant in recent literature [Bala et al. 1999; Chen et al. 2000; Deaver et al. 1999; Gschwind et al. 2000; Klaiber 2000; Zheng and Thompson 2000]. The structure of these is fairly similar:

(1) *Profiled execution*. The simulator's effort should be concentrated on "hot" areas of code that are frequently executed. For example, initialization code that is executed only once should not be translated or optimized. To determine which execution paths are hot, the source program is executed in some manner and profile information is gathered. Time invested in doing this is assumed to be recouped eventually.

When source and target architectures are dissimilar, or the source architecture is uncomplicated (such as a reduced instruction set computer (RISC) processor) then interpretation of the source program is typically employed to execute the source program [Bala et al. 1999; Gschwind et al. 2000; Transmeta Corporation 2001; Zheng and Thompson 2000]. The alternative approach, direct execution, is best summed up by Rosenblum et al. [1995, p. 36]:

> By far the fastest simulator of the CPU, MMU, and memory system of an SGI multiprocessor is an SGI multiprocessor.

In other words, when the source and target architectures are the same, as in the case where the goal is dynamic optimization of a source program, the source program can be executed directly by the central processing unit (CPU). The simulator regains control periodically as a result of appropriately modifying the source program [Chen et al. 2000] or by less direct means such as interrupts [Gorton 2001].

(2) *Hot path detection*. In lieu of hardware support, hot paths may be detected by keeping counters to record frequency of execution [Zheng and Thompson 2000], or by watching for code that is structurally likely to be hot, like the target of a backward branch [Bala et al. 1999]. With hardware support, the program's program counter can be sampled at intervals to detect hot spots [Deaver et al. 1999].

Some other considerations are that paths may be strategically *excluded* if they are too expensive or difficult to translate [Zheng and Thompson 2000], and choosing good stopping points for paths can be as important as choosing good starting points in terms

of keeping a manageable number of traces [Gschwind et al. 2000].

(3) *Code generation and optimization*. Once a hot path has been noted, the simulator will translate it into code for the target architecture, or perhaps optimize the code. The correctness of the translation is always at issue, and some empirical verification techniques are discussed in [Zheng and Thompson 2000].

(4) *"Bail-out" mechanism*. In the case of dynamic optimization systems (where the source and target architectures are the same), there is the potential for a negative impact on the source program's performance. A bail-out mechanism [Bala et al. 1999] heuristically tries to detect such a problem and revert back to the source program's direct execution; this can be spotted, for example, by monitoring the stability of the working set of paths. Such a mechanism can also be used to avoid handling complicated cases.

Another recurring theme in recent binary translation work is the issue of hardware support for binary translation, especially for translating code for legacy architectures into VLIW code. This has attracted interest because VLIW architectures promise legacy architecture implementations which have higher performance, greater instruction-level parallelism [Ebcioğlu and Altman 1996, 1997], higher clock rates [Altman et al. 2000a; Gschwind et al. 2000], and lower power requirements [Klaiber 2000]. Binary translation work in these processors is still done by software at run-time, and is thus still dynamic binary translation, although occasionally packaged under more fanciful names to enrapture venture capitalists [Geppert and Perry 2000]. The key idea in these systems is that, for efficiency, the target VLIW should provide a superset of the source architecture [Ebcioğlu and Altman 1997]; these extra resources, unseen by the source program, can be used by the binary translator for aggressive optimizations or to simulate troublesome aspects of the source architecture.

2.14. Java

Java is implemented by static compilation to bytecode instructions for the Java virtual machine, or JVM. Early JVMs were only interpreters, resulting in less-than-stellar performance:

> Interpreting bytecodes is slow. [Cramer et al. 1997, p. 37]

> Java isn't just slow, it's *really* slow, *surprisingly* slow. [Tyma 1998, p. 41]

Regardless of how vitriolic the expression, the message was that Java programs had to run faster, and the primary means looked to for accomplishing this was JIT compilation of Java bytecodes. Indeed, Java brought the term *just-in-time* into common use in computing literature.[13] Unquestionably, the pressure for fast Java implementations spurred a renaissance in JIT research; at no other time in history has such concentrated time and money been invested in it.

An early view of Java JIT compilation was given by Cramer et al. [1997], who were engineers at Sun Microsystems, the progenitor of Java. They made the observation that there is an upper bound on the speedup achievable by JIT compilation, noting that interpretation proper only accounted for 68% of execution time in a profile they ran. They also advocated the direct use of JVM bytecodes, a stack-based instruction set, as an intermediate representation for JIT compilation and optimization. In retrospect, this is a minority viewpoint; most later work, including Sun's own [Sun Microsystems 2001], invariably began by converting JVM code into a register-based intermediate representation.

The interesting trend in Java JIT work [Adl-Tabatabai et al. 1998; Bik et al. 1999; Burke et al. 1999; Cierniak and Li 1997; Ishizaki et al. 1999; Krall and Grafl 1997; Krall 1998; Yang et al. 1999] is the implicit assumption that mere

[13] Gosling [2001] pointed out that the term *just-in-time* was borrowed from manufacturing terminology, and traced his own use of the term back to about 1993.

translation from bytecode to native code is not enough: code optimization is necessary too. At the same time, this work recognizes that traditional optimization techniques are expensive, and looks for modifications to optimization algorithms that strike a balance between speed of algorithm execution and speed of the resulting code.

There have also been approaches to Java JIT compilation besides the usual interpret-first-optimize-later. A compile-only strategy, with no interpreter whatsoever, was adopted by Burke et al. [1999], who also implemented their system in Java; improvements to their JIT directly benefited their system. Agesen [1997] translated JVM bytecodes into Self code, to leverage optimizations already existing in the Self compiler. Annotations were tried by Azevedo et al. [1999] to shift the effort of code optimization prior to runtime: information needed for efficient JIT optimization was precomputed and tagged on to bytecode as annotations, which were then used by the JIT system to assist its work. Finally, Plezbert and Cytron [1997] proposed and evaluated the idea of "continuous compilation" for Java in which an interpreter and compiler would execute concurrently, preferably on separate processors.[14]

3. CLASSIFICATION OF JIT SYSTEMS

In the course of surveying JIT work, some common attributes emerged. We propose that JIT systems can be classified according to three properties:

(1) *Invocation*. A JIT compiler is explicitly invoked if the user must take some action to cause compilation at runtime. An implicitly invoked JIT compiler is transparent to the user.

(2) *Executability*. JIT systems typically involve two languages: a source language to translate from, and a target language to translate to (although these languages can be the same, if the JIT system is only performing optimization on-the-fly). We call a JIT system *monoexecutable* if it can only execute one of these languages, and *polyexecutable* if can execute more than one. Polyexecutable JIT systems have the luxury of deciding when compiler invocation is warranted, since either program representation can be used.

(3) *Concurrency*. This property characterizes how the JIT compiler executes, relative to the program itself. If program execution pauses under its own volition to permit compilation, it is not concurrent; the JIT compiler in this case may be invoked via subroutine call, message transmission, or transfer of control to a coroutine. In contrast, a concurrent JIT compiler can operate as the program executes concurrently: in a separate thread or process, even on a different processor.

JIT systems that function in hard real time may constitute a fourth classifying property, but there seems to be little research in the area at present; it is unclear if hard real-time constraints pose any unique problems to JIT systems.

Some trends are apparent. For instance, implicitly invoked JIT compilers are definitely predominant in recent work. Executability varies from system to system, but this is more an issue of design than an issue of JIT technology. Work on concurrent JIT compilers is currently only beginning, and will likely increase in importance as processor technology evolves.

4. TOOLS FOR JIT COMPILATION

General, portable tools for JIT compilation that help with the dynamic generation of binary code did not appear until relatively recently. To varying degrees, these toolkits address three issues:

(1) *Binary code generation*. As argued in Ramsey and Fernández [1995], emitting binary code such as machine language is a situation rife with opportunities for error. There are associated

[14] As opposed to the ongoing optimization of Kistler's [2001] "continuous optimization," only compilation occurred concurrently using "continuous compilation," and only happened once.

Table 1. Comparison of JIT Toolkits

Source	Binary code generation	Cache coherence	Execution	Abstract interface	Input
Engler [1996]	•	•	•	•	ad hoc
Engler and Proebsting [1994]	•	•	•	•	tree
Fraser and Proebsting [1999]	•	•	•	•	postfix
Keppel [1991]		•	•	•	n/a
Ramsey and Fernández [1995]	•				ad hoc

Note: n/a = not applicable.

bookkeeping tasks too: information may not yet be available upon initial code generation, like the location of forward branch targets. Once discovered, the information must be backpatched into the appropriate locations.

(2) *Cache coherence*. CPU speed advances have far outstripped memory speed advances in recent years [Hennessy and Patterson 1996]. To compensate, modern CPUs incorporate a small, fast cache memory, the contents of which may get temporarily out of sync with main memory. When dynamically generating code, care must be taken to ensure that the cache contents reflect code written to main memory before execution is attempted. The situation is even more complicated when several CPUs share a single memory. Keppel [1991] gave a detailed discussion.

(3) *Execution*. The hardware or operating system may impose restrictions which limit where executable code may reside. For example, memory earmarked for data may not allow execution (i.e., instruction fetches) by default, meaning that code could be generated into the data memory, but not executed without platform-specific wrangling. Again, refer to Keppel [1991].

Only the first issue is relevant for JIT compilation to interpreted virtual machine code—interpreters don't directly execute the code they interpret—but there is no reason why JIT compilation tools cannot be useful for generation of nonnative code as well.

Table I gives a comparison of the toolkits. In addition to indicating how well the toolkits support the three areas above, we have added two extra categories. First, an *abstract interface* is one that is architecture-independent. Use of a toolkit's abstract interface implies that very little, if any, of the user's code needs modification in order to use a new platform. The drawbacks are that architecture-dependent operations like register allocation may be difficult, and the mapping from abstract to actual machine may be suboptimal, such as a mapping from RISC abstraction to complex instruction set computer (CISC) machinery.

Second, *input* refers to the structure, if any, of the input expected by the toolkit. With respect to JIT compilation, more complicated input structures take more time and space for the user to produce and the toolkit to consume [Engler 1996].

Using a tool may solve some problems but introduce others. Tools for binary code generation help avoid many errors compared to manually emitting binary code. These tools, however, require detailed knowledge of binary instruction formats whose specification may itself be prone to error. Engler and Hsieh [2000] presented a "metatool" that can automatically derive these instruction encodings by repeatedly querying the existing system assembler with varying inputs.

5. CONCLUSION

Dynamic, or just-in-time, compilation is an old implementation technique with a fragmented history. By collecting this historical information together, we hope to shorten the voyage of rediscovery.

ACKNOWLEDGMENTS

Thanks to Nigel Horspool, Shannon Jaeger, and Mike Zastre, who proofread and commented on drafts of this paper. Comments from the anonymous referees helped improve the presentation as well. Also, thanks to Rick Gorton, James Gosling, Thomas Kistler, Ralph Mauriello, and Jim Mitchell for supplying historical information and clarifications. Evelyn Duesterwald's PLDI 2000 tutorial notes were helpful in preparing Section 2.9.

REFERENCES

ABRAMS, P. S. 1970. An APL machine. Ph.D. dissertation. Stanford University, Stanford, CA. Also, Stanford Linear Accelerator Center (SLAC) Rep. 114.

ADL-TABATABAI, A.-R., CIERNIAK, M., LUEH, G.-Y., PARIKH, V. M., AND STICHNOTH, J. M. 1998. Fast, effective code generation in a just-in-time Java compiler. In *PLDI '98*. 280–290.

AGESEN, O. 1996. Concrete type inference: Delivering object-oriented applications. Ph.D. dissertation. Stanford University, Stanford, CA. Also Tech. Rep. SMLI TR-96-52, Sun Microsystems, Santa Clara, CA (Jan. 1996).

AGESEN, O. 1997. Design and implementation of Pep, a Java just-in-time translator. *Theor. Prac. Obj. Syst. 3*, 2, 127–155.

AGESEN, O. AND HÖLZLE, U. 1995. Type feedback vs. concrete type inference: A comparison of optimization techniques for object-oriented languages. In *Proceedings of OOPSLA '95*. 91–107.

ALTMAN, E., GSCHWIND, M., SATHAYE, S., KOSONOCKY, S., BRIGHT, A., FRITTS, J., LEDAK, P., APPENZELLER, D., AGRICOLA, C., AND FILAN, Z. 2000a. BOA: The architecture of a binary translation processor. Tech. Rep. RC 21665, IBM Research Division, Yorktown Heights, NY.

ALTMAN, E. R., KAELI, D., AND SHEFFER, Y. 2000b. Welcome to the opportunities of binary translation. *IEEE Comput. 33*, 3 (March), 40–45.

ARMSTRONG, J. 1997. The development of Erlang. In *Proceedings of ICFP '97* (1997). 196–203.

AUSLANDER, J., PHILIPOSE, M., CHAMBERS, C., EGGERS, S. J., AND BERSHAD, B. N. 1996. Fast, effective dynamic compilation. In *Proceedings of PLDI '96*. 149–159.

AZEVEDO, A., NICOLAU, A., AND HUMMEL, J. 1999. Java annotation-aware just-in-time (AJIT) compilation system. In *Proceedings of JAVA '99*. 142–151.

BALA, V., DUESTERWALD, E., AND BANERJIA, S. 1999. Transparent dynamic optimization. Tech. Rep. HPL-1999-77, Hewlett-Packard, Polo Alto, CA.

BARTLETT, J. 1992. *Familiar Quotations* (16th ed.). J. Kaplan, Ed. Little, Brown and Company, Boston, MA.

BELL, J. R. 1973. Threaded code. *Commun. ACM 16*, 6 (June), 370–372.

BENTLEY, J. 1988. Little languages. In *More Programming Pearls*. Addison-Wesley, Reading, MA, 83–100.

BIK, A. J. C., GIRKAR, M., AND HAGHIGHAT, M. R. 1999. Experiences with Java JIT optimization. In *Innovative Architecture for Future Generation High-Performance Processors and Systems*. IEEE Computer Society Press, Los Alamitos, CA, 87–94.

BROWN, P. J. 1976. Throw-away compiling. *Softw.—Pract. Exp. 6*, 423–434.

BROWN, P. J. 1990. *Writing Interactive Compilers and Interpreters*. Wiley, New York, NY.

BURGER, R. G. 1997. Efficient compilation and profile-driven dynamic recompilation in scheme. Ph.D. dissertation, Indiana University, Bloomington, IN.

BURKE, M. G., CHOI, J.-D., FINK, S., GROVE, D., HIND, M., SARKAR, V., SERRANO, M. J., SREEDHAR, V. C., AND SRINIVASAN, H. 1999. The Jalapeño dynamic optimizing compiler for Java. In *Proceedings of JAVA '99*. 129–141.

CARDELLI, L. 1984. Compiling a functional language. In *1984 Symposium on Lisp and Functional Programming*. 208–217.

CHAMBERS, C. 1992. The design and implementation of the self compiler, an optimizing compiler for object-oriented programming languages. Ph.D. dissertation. Stanford University, Stanford, CA.

CHAMBERS, C. AND UNGAR, D. 1989. Customization: optimizing compiler technology for Self, a dynamically-typed object-oriented programming language. In *Proceedings of PLDI '89*. 146–160.

CHAMBERS, C. AND UNGAR, D. 1990. Iterative type analysis and extended message splitting: Optimizing dynamically-typed object-oriented programs. In *Proceedings of PLDI '90*. 150–164.

CHAMBERS, C. AND UNGAR, D. 1991. Making pure object-oriented languages practical. In *Proceedings of OOPSLA '91*. 1–15.

CHAMBERS, C., UNGAR, D., AND LEE, E. 1989. An efficient implementation of Self, a dynamically-typed object-oriented programming language based on prototypes. In *Proceedings of OOPSLA '89*. 49–70.

CHEN, W.-K., LERNER, S., CHAIKEN, R., AND GILLIES, D. M. 2000. Mojo: a dynamic optimization system. In *Proceedings of the Third ACM Workshop on Feedback-Directed and Dynamic Optimization* (FDDO-3, Dec. 2000).

CIERNIAK, M. AND LI, W. 1997. Briki: an optimizing Java compiler. In *Proceedings of IEEE COMPCON '97*. 179–184.

CMELIK, B. AND KEPPEL, D. 1994. Shade: A fast instruction-set simulator for execution profiling. In *Proceedings of the 1994 Conference on Measurement and Modeling of Computer Systems*. 128–137.

CONSEL, C., HORNOF, L., MARLET, R., MULLER, G., THIBAULT, S., VOLANSCHI, E.-N., LAWALL, J., AND NOYÉ, J. 1998. Tempo: Specializing systems applications and beyond. *ACM Comput. Surv. 30*, 3 (Sept.), 5pp.

CONSEL, C. AND NOËL, F. 1996. A general approach for run-time specialization and its application to C. In *Proceedings of POPL '96.* 145–156.

CRAMER, T., FRIEDMAN, R., MILLER, T., SEBERGER, D., WILSON, R., AND WOLCZKO, M. 1997. Compiling Java just in time. *IEEE Micro 17*, 3 (May/June), 36–43.

DAKIN, R. J. AND POOLE, P. C. 1973. A mixed code approach. *The Comput. J. 16*, 3, 219–222.

DAWSON, J. L. 1973. Combining interpretive code with machine code. *The Comput. J. 16*, 3, 216–219.

DEAVER, D., GORTON, R., AND RUBIN, N. 1999. Wiggins/Redstone: An on-line program specializer. In *Proceedings of the IEEE Hot Chips XI Conference* (Aug. 1999). IEEE Computer Society Press, Los, Alamitos, CA.

DEUTSCH, L. P. AND SCHIFFMAN, A. M. 1984. Efficient implementation of the Smalltalk-80 system. In *Proceedings of POPL '84*. 297–302.

DIECKMANN, S. AND HÖLZLE, U. 1997. The space overhead of customization. Tech. Rep. TRCS 97-21. University of California, Santa Barbara, Santa Barbara, CA.

EBCIOĞLU, K. AND ALTMAN, E. R. 1996. DAISY: Dynamic compilation for 100% architectural compatibility. Tech. Rep. RC 20538. IBM Research Division, Yorktown Heights, NY.

EBCIOĞLU, K. AND ALTMAN, E. R. 1997. Daisy: Dynamic compilation for 100% architectural compatibility. In *Proceedings of ISCA '97*. 26–37.

ENGLER, D. R. 1996. VCODE: a retargetable, extensible, very fast dynamic code generation system. In *Proceedings of PLDI '96.* 160–170.

ENGLER, D. R. AND HSIEH, W. C. 2000. DERIVE: A tool that automatically reverse-engineers instruction encodings. In *Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization* (Dynamo '00). 12–22.

ENGLER, D. R., HSIEH, W. C., AND KAASHOEK, M. F. 1996. C: A language for high-level, efficient, and machine-independent dynamic code generation. In *Proceedings of POPL '96.* 131–144.

ENGLER, D. R. AND PROEBSTING, T. A. 1994. DCG: An efficient, retargetable dynamic code generation system. In *Proceedings of ASPLOS VI.* 263–272.

FRANZ, M. 1994. *Code-generation on-the-fly: A key to portable software*. Ph.D. dissertation. ETH Zurich, Zurich, Switzerland.

FRANZ, M. AND KISTLER, T. 1997. Slim binaries. *Commun. ACM 40*, 12 (Dec.), 87–94.

FRASER, C. W. AND PROEBSTING, T. A. 1999. Finite-state code generation. In *Proceedings of PLDI '99*. 270–280.

GABRIEL, R. P. AND MASINTER, L. M. 1985. *Performance and Evaluation of Lisp Systems*. MIT Press, Cambridge, MA.

GEPPERT, L. AND PERRY, T. S. 2000. Transmeta's magic show. *IEEE Spectr. 37*, 5 (May), 26–33.

GOLDBERG, A. AND ROBSON, D. 1985. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, MA.

GORTON, R. 2001. Private communication.

GOSLING, J. 2001. Private communication.

GSCHWIND, M., ALTMAN, E. R., SATHAYE, S., LEDAK, P., AND APPENZELLER, D. 2000. Dynamic and transparent binary translation. *IEEE Comput. 33*, 3, 54–59.

HAMMOND, J. 1977. BASIC—an evaluation of processing methods and a study of some programs. *Softw.—Pract. Exp. 7*, 697–711.

HANSEN, G. J. 1974. Adaptive systems for the dynamic run-time optimization of programs. Ph.D. dissertation. Carnegie-Mellon University, Pittsburgh, PA.

HAYGOOD, R. C. 1994. Native code compilation in SICStus Prolog. In *Proceedings of the Eleventh International Conference on Logic Programming*. 190–204.

HENNESSY, J. L. AND PATTERSON, D. A. 1996. *Computer Architecture: A Quantitative Approach*, 2nd ed. Morgan Kaufmann, San Francisco, CA.

HÖLZLE, U. 1994. *Adaptive optimization for Self: Reconciling high performance with exploratory programming*. Ph.D. dissertation. Carnegie-Mellon University, Pittsburgh, PA.

HÖLZLE, U. AND UNGAR, D. 1994a. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proceedings of PLDI '94.* 326–336.

HÖLZLE, U. AND UNGAR, D. 1994b. A third-generation Self implementation: Reconciling responsiveness with performance. In *Proceedings of OOPSLA '94.* 229–243.

ISHIZAKI, K., KAWAHITO, M., YASUE, T., TAKEUCHI, M., OGASAWARA, T., SUGANUMA, T., ONODERA, T., KOMATSU, H., AND NAKATANI, T. 1999. Design, implementation, and evaluation of optimizations in a just-in-time compiler. In *Proceedings of JAVA '99.* 119–128.

JOHANSSON, E., PETTERSSON, M., AND SAGONAS, K. 2000. A high performance Erlang system. In *Proceedings of PPDP '00.* 32–43.

JOHNSTON, R. L. 1977. The dynamic incremental compiler of APL\3000. In *APL '79 Conference Proceedings.* Published in *APL Quote Quad 9*, 4 (June), Pt. 1, 82–87.

JONES, N. D., GOMARD, C. K., AND SESTOFT, P. 1993. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, Englewood Cliffs, NJ.

KEPPEL, D. 1991. A portable interface for on-the-fly instruction space modification. In *Proceedings of ASPLOS IV.* 86–95.

KEPPEL, D., EGGERS, S. J., AND HENRY, R. R. 1991. A case for runtime code generation. Tech. Rep.

91-11-04. Department of Computer Science and Engineering, University of Washington, Seattle, WA.

KISTLER, T. 1997. Dynamic runtime optimization. In *Proceedings of the Joint Modular Languages Conference* (JMLC '97). 53–66.

KISTLER, T. 1999. *Continuous program optimization*. Ph.D. dissertation. University of California, Irvine, Irvine, CA.

KISTLER, T. 2001. Private communication.

KISTLER, T. AND FRANZ, M. 1999. The case for dynamic optimization: Improving memory-hierarchy performance by continuously adapting the internal storage layout of heap objects at run-time. Tech. Rep. 99-21 (May). University of California, Irvine, Irvine, CA. Revised September, 1999.

KLAIBER, A. 2000. The technology behind Crusoe processors. Tech. Rep. (Jan.), Transmeta Corporation, Santa Clara, CA.

KNUTH, D. E. 1971. An empirical study of Fortran programs. *Softw.—Pract. Exp. 1*, 105–133.

KRALL, A. 1998. Efficient JavaVM just-in-time compilation. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques* (PACT '98). 205–212.

KRALL, A. AND GRAFL, R. 1997. A Java just-in-time compiler that transcends JavaVM's 32 bit barrier. In *Proceedings of PPoPP '97 Workshop on Java for Science and Engineering.*

LEE, P. AND LEONE, M. 1996. Optimizing ML with run-time code generation. In *Proceedings of PLDI '96.* 137–148.

LEE, S., YANG, B.-S., KIM, S., PARK, S., MOON, S.-M., EBCIOĞLU, K., AND ALTMAN, E. 2000. Efficient Java exception handling in just-in-time compilation. In *Proceedings of Java 2000.* 1–8.

LEONE, M. AND DYBVIG, R. K. 1997. Dynamo: A staged compiler architecture for dynamic program optimization. Tech. Rep. 490. Computer Science Department, Indiana University, Bloomington, IN.

LEONE, M. AND LEE, P. 1994. Lightweight run-time code generation. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*. 97–106.

MARLET, R., CONSEL, C., AND BOINOT, P. 1999. Efficient incremental run-time specialization for free. In *PLDI '99.* 281–292.

MAURIELLO, R. 2000. Private communication.

MAY, C. 1987. Mimic: A fast System/370 simulator. In *Proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques* (June). ACM Press, New York, NY, 1–13.

MCCARTHY, J. 1960. Recursive functions of symbolic expressions and their computation by machine, part I. *Commun. ACM 3*, 4, 184–195.

MCCARTHY, J. 1981. History of LISP. In *History of Programming Languages*, R. L. Wexelblat, Ed. Academic Press, New York, NY, 173–185.

MILLER, T. C. 1977. Tentative compilation: A design for an APL compiler. In *APL '79 Conference Proceedings.* Volume 9 Published in *APL Quote Quad 9*, 4 (June), Pt. 1, 88–95.

MITCHELL, J. G. 1970. The design and construction of flexible and efficient interactive programming systems. Ph.D. dissertation. Carnegie-Mellon University, Pittsburgh, PA.

MITCHELL, J. G. 2000. Private communication.

MITCHELL, J. G., PERLIS, A. J., AND VAN ZOEREN, H. R. 1968. LC^2: A language for conversational computing. In *Interactive Systems for Experimental Applied Mathematics*, M. Klerer and J. Reinfelds, Eds. Academic Press, New York, NY. (Proceedings of 1967 ACM Symposium.)

MOCK, M., BERRYMAN, M., CHAMBERS, C., AND EGGERS, S. J. 1999. Calpa: A tool for automating dynamic compilation. In *Proceedings of the Second ACM Workshop on Feedback-Directed and Dynamic Optimization.* 100–109.

NG, T. S. AND CANTONI, A. 1976. Run time interaction with FORTRAN using mixed code. *The Comput. J. 19*, 1, 91–92.

PITTMAN, T. 1987. Two-level hybrid interpreter/native code execution for combined space-time program efficiency. In *Proceedings of the SIGPLAN Symposium on Interpreters and Interpretive Techniques*. ACM Press, New York, NY, 150–152.

PIUMARTA, I. AND RICCARDI, F. 1998. Optimizing direct threaded code by selective inlining. In *Proceedings of PLDI '98.* 291–300.

PLEZBERT, M. P. AND CYTRON, R. K. 1997. Does "just in time" = "better late then never"? In *Proceedings of POPL '97.* 120–131.

POLETTO, M., ENGLER, D. R., AND KAASHOEK, M. F. 1997. tcc: A system for fast, flexible, and high-level dynamic code generation. In *Proceedings of PLDI '97.* 109–121.

RAMSEY, N. AND FERNÁNDEZ, M. 1995. The New Jersey machine-code toolkit. In *Proceedings of the 1995 USENIX Technical Conference.* 289–302.

RAU, B. R. 1978. Levels of representation of programs and the architecture of universal host machines. In *Proceedings of the 11th Annual Microprogramming Workshop* (MICRO-11). 67–79.

RÉMY, D., LEROY, X., AND WEIS, P. 1999. Objective Caml—a general purpose high-level programming language. *ERCIM News 36*, 29–30.

ROSENBLUM, M., HERROD, S. A., WITCHEL, E., AND GUPTA, A. 1995. Complete computer system simulation: The SimOS approach. *IEEE Parall. Distrib. Tech. 3*, 4 (Winter), 34–43.

SCHROEDER, S. C. AND VAUGHN, L. E. 1973. A high order language optimal execution processor: *F*ast *I*ntent *R*ecognition *Syst*em (FIRST). In *Proceedings of a Symposium on High-Level-Language*

Computer Architecture. Published in *SIGPLAN 8*, 11 (Nov.), 109–116.

SEBESTA, R. W. 1999. *Concepts of Programming Languages* (4th ed.). Addison-Wesley, Reading, MA.

SMITH, R. B. AND UNGAR, D. 1995. Programming as an experience: The inspiration for Self. In *Proceedings of ECOOP '95*.

SUN MICROSYSTEMS. 2001. The Java HotSpot virtual machine. White paper. Sun Microsystems, Santa Clara, CA.

THIBAULT, S., CONSEL, C., LAWALL, J. L., MARLET, R., AND MULLER, G. 2000. Static and dynamic program compilation by interpreter specialization. *Higher-Order Symbol. Computat. 13*, 161–178.

THOMPSON, K. 1968. Regular expression search algorithm. *Commun. ACM 11*, 6 (June), 419–422.

TRANSMETA CORPORATION. 2001. Code morphing software. Available online at http://www.transmeta.com/echnology/architecture/code_morphing.html. Transmeta Corporation, Santa Clara, CA.

TYMA, P. 1998. Why are we using Java again? *Commun. ACM 41*, 6, 38–42.

UNG, D. AND CIFUENTES, C. 2000. Machine-adaptable dynamic binary translation. In *Proceedings of Dynamo '00*. 41–51.

UNGAR, D. AND SMITH, R. B. 1987. Self: The power of simplicity. In *Proceedings of OOPSLA '87*. 227–242.

UNGAR, D., SMITH, R. B., CHAMBERS, C., AND HÖLZLE, U. 1992. Object, message, and performance: How they coexist in Self. *IEEE Comput. 25*, 10 (Oct.), 53–64.

UNIVERSITY OF MICHIGAN. 1966a. The System Loader. In *University of Michigan Executive System for the IBM 7090 Computer*, Vol. 1. University of Michigan, Ann Arbor, MI.

UNIVERSITY OF MICHIGAN. 1966b. The "University of Michigan Assembly Program" ("UMAP"). In *University of Michigan Executive System for the IBM 7090 Computer*, Vol. 2. University of Michigan, Ann Arbor, MI.

VAN DYKE, E. J. 1977. A dynamic incremental compiler for an interpretive language. *Hewlett-Packard J. 28*, 11 (July), 17–24.

VAN ROY, P. 1994. The wonder years of sequential Prolog implementation. *J. Logic Program. 19–20*, 385–441.

WICKLINE, P., LEE, P., AND PFENNING, F. 1998. Run-time code generation and Modal-ML. In *Proceedings of PLDI '98*. 224–235.

WIRTH, N. AND GUTKNECHT, J. 1989. The Oberon system. *Softw.—Pract. Exp. 19*, 9 (Sep.), 857–893.

YANG, B.-S., MOON, S.-M., PARK, S., LEE, J., LEE, S., PARK, J., CHUNG, Y. C., KIM, S., EBCIOĞLU, K., AND ALTMAN, E. 1999. LaTTe: A Java VM just-in-time compiler with fast and efficient register allocation. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. 128–138. IEEE Computer Society Press, Los Alamitos, CA.

ZHENG, C. AND THOMPSON, C. 2000. PA-RISC to IA-64: Transparent execution, no recompilation. *IEEE Comput. 33*, 3 (March), 47–52.

Received July 2002; revised March 2003; accepted February 2003

EXHIBIT I

MORRISON & FOERSTER LLP
MICHAEL A. JACOBS (Bar No. 111664)
mjacobs@mofo.com
MARC DAVID PETERS (Bar No. 211725)
mdpeters@mofo.com
755 Page Mill Road
Palo Alto, CA 94304-1018
Telephone: (650) 813-5600 / Facsimile: (650) 494-0792

BOIES, SCHILLER & FLEXNER LLP
DAVID BOIES (Admitted *Pro Hac Vice*)
dboies@bsfllp.com
333 Main Street
Armonk, NY 10504
Telephone: (914) 749-8200 / Facsimile: (914) 749-8300
STEVEN C. HOLTZMAN (Bar No. 144177)
sholtzman@bsfllp.com
1999 Harrison St., Suite 900
Oakland, CA 94612
Telephone: (510) 874-1000 / Facsimile: (510) 874-1460

ORACLE CORPORATION
DORIAN DALEY (Bar No. 129049)
dorian.daley@oracle.com
DEBORAH K. MILLER (Bar No. 95527)
deborah.miller@oracle.com
MATTHEW M. SARBORARIA (Bar No. 211600)
matthew.sarboraria@oracle.com
500 Oracle Parkway
Redwood City, CA 94065
Telephone: (650) 506-5200 / Facsimile: (650) 506-7114

Attorneys for Plaintiff
ORACLE AMERICA, INC.

UNITED STATES DISTRICT COURT

NORTHERN DISTRICT OF CALIFORNIA

SAN FRANCISCO DIVISION

ORACLE AMERICA, INC.

Plaintiff,

v.

GOOGLE INC.

Defendant.

Case No. CV 10-03561 WHA

PLAINTIFF'S RESPONSES AND OBJECTIONS TO DEFENDANT GOOGLE INC.'S FIRST SET OF INTERROGATORIES TO PLAINTIFF ORACLE AMERICA, INC. (NOS. 1-10)

Dept.: Courtroom 9, 19th Floor
Judge: Honorable William H. Alsup

INTERROGATORY NO. 8:

Identify with specificity all portions of the Java documentation that were automatically generated using software and explain how each was generated.

RESPONSE TO INTERROGATORY NO. 8:

Generally, all Java API documentation is automatically generated using the Javadoc software tool. Javadoc is a documentation generator developed by Sun Microsystems. Javadoc is used to generate API documentation in HTML format from Java source code, based on standardized tags and comments written by source code programmers. A Javadoc comment is set off from source code by comment tags "/**" and "*/". For example, the first paragraph in such a comment may be a description of the method documented. Next, certain tags are used to signify certain information (*e.g.*, @param name description describes a method parameter, @return description describes a method return value, @throws describes an exception the method may throw).

Discovery is ongoing, and Oracle has not yet completed its investigation of the documents and facts relevant to the claims and defenses asserted in this action. Accordingly, Oracle's responses are based on the information reasonably available at this time and Oracle will supplement this response as appropriate under the Federal Rules of Civil Procedure.

INTERROGATORY NO. 9:

State in detail the terms of a fair, reasonable and non-discriminatory license to Oracle's TCK consistent with Oracle's obligations under the Java Specification Participation Agreement, including the bases of any computation of any monetary elements of such a license and an explanation of why such a license is fair, reasonable and non-discriminatory.

RESPONSE TO INTERROGATORY NO. 9:

The JSPA permits a Specification Lead to impose terms and conditions as part of a TCK license. Any interested party may license the Spec Lead's TCK under "non-discriminatory, fair and reasonable" terms and conditions and "such terms and conditions shall be determined by the Spec Lead in its reasonable discretion." (JSPA, § 5.F.I.) Oracle's TCK licenses comport with its

obligations under the JSPA, and, with respect to the terms of its TCK licenses, Oracle directs Google to its TCK licenses produced in this action pursuant to Fed. R. Civ. P. 33(d).

As for the terms of any TCK license to Android, none has ever been requested, and Oracle accordingly has never considered what reasonable terms or royalty computation of one might be. Issuing a TCK license to Android makes no sense, given that Android does not implement the entire Java specification and is accordingly not compliant.

In addition to the general objections stated above, Oracle further objects to this interrogatory insofar as it seeks information protected from discovery by the attorney-client privilege or the attorney work-product doctrine. Oracle further objects to this request on the grounds that determinations of why or whether the terms of any license are fair, reasonable, and non-discriminatory are purely matters of legal opinion and are therefore not within the scope of inquiry permitted by Fed. R. Civ. P. 33(a)(2). Discovery is ongoing, and Oracle has not yet completed its investigation of the documents and facts relevant to the claims and defenses asserted in this action. Accordingly, Oracle's responses are based on the information reasonably available at this time and Oracle will supplement this response as appropriate under the Federal Rules of Civil Procedure.

INTERROGATORY NO. 10:

State in detail Oracle's factual bases for its allegation that the doctrine of assignor estoppel bars Google from challenging the validity of each of the patents-in-suit to which Oracle contends the doctrine applies.

RESPONSE TO INTERROGATORY NO. 10:

Assignor estoppel bars Google from challenging the validity of any patent assigned by an inventor with whom Google is in privity. Google hired named inventors of Oracle's patents—including at least Frank Yellin, co-inventor of the '520 patent, and Lars Bak and Robert Griesemer, co-inventors of the '205 patent—to work on Java and Web browser technologies. Google is in the best position to know how it availed itself of the inventors' knowledge and assistance. As the inventors' employer, it is Google, not Oracle that possesses detailed

EXHIBIT J

8/30/10 N.Y. Times B1
2010 WLNR 17231951

New York Times (NY)

August 30, 2010

Section: B

Software War Pits Oracle Vs. Google

STEVE LOHR

Software war builds following Oracle's filing of lawsuit against Google, accusing it of copyright and patent infringement for its Android operating system for smartphones and other mobile devices; Oracle claims Google is using ideas and code from Java software tools initially developed by Sun Microsystems in 1995; Sun was purchased by Oracle in January; Google says software was built without using Sun's intellectual property, instead claiming that Oracle seeks to re-establish corporate control of open-sourced Java technology; photo (M)

Free open-source software began with high-technology tinkerers and researchers. Sharing code and ideas was their priority, not profits. In the tech industry, they were sometimes compared to socialists and communists.

Those days are long gone. Some of the communal idealism remains, but as open-source software is used more by big technology companies including I.B.M., Oracle, Hewlett-Packard, Google and Apple -- even Microsoft -- it has also become a weapon in corporate warfare.

An unusually public salvo came this month, when Oracle sued Google, accusing it of copyright and patent infringement. Oracle claims that Google's Android operating system for smartphones and other mobile devices is illegally using ideas and code from Java, a set of software tools initially developed by Sun Microsystems in 1995. Oracle bought Sun in January,

Google denies the charges against Android, which is also open-source software, saying that it built the operating system and its own Java tools without using Sun's intellectual property.

Google instead sees the suit as a move by Oracle to re-establish corporate control of Java, something that Sun's executives were reluctant to do. "This action is not against Android per se but against any Java development not sanctioned by Oracle," said Kent Walker, general counsel of Google. "The lawsuit is trying to put the genie back in the bottle."

With open-source software, programmers can view the underlying source code and make modifications and fix bugs, as long as they abide by certain rules. Open-source programs are typically distributed free.

An estimated three-quarters of all open-source software is chugging away in service of the profit-seeking corporate world. It is used, in the form of the Linux operating system or the Apache Web server, to run data centers that power the Web.

Every company deploys open source differently as a tool to cut costs or as a weapon to gain an advantage over rivals.

The corporate battles are fought with software programmers who contribute to open-source programs, in the marketplace with sales campaigns and in standards bodies that govern open-source projects. The Oracle-Google clash is the exceptional case that ended up in court.

Their confrontation, according to Douglas Lea, a computer scientist at the State University of New York at Oswego, is a new front in what he calls "the open-source proxy wars," in which big companies use open source to gain an upper hand in the commercial marketplace.

"It's not so much good companies and bad companies in this kind of situation," said Mr. Lea, a member of the executive committee of the Java Community Process, a group that defines Java features and standards. "These companies compete viciously and have different interests. And in this case, you have two corporations that champion different forms of open source."

The roots of the Oracle suit go back well before Oracle acquired Sun. After Sun made Java open source in 2006 to broaden its adoption, its strategy was to let developers and companies freely use the Java technology deployed in data centers. **Google** was a major participant in **contributing** features and shaping standards for this so-called big **Java** in the **Java** Community Process, where Sun (now Oracle) retains the status of first among equals.

But Sun decided it would make money in the fast-growing field of cellphones with a set of software tools tailored for that market, called Java Micro Edition. This "small Java" is free for most developers, but Sun negotiates commercial licenses for big companies that want to make their own products. Licensees include Nokia, Research In Motion, Motorola, LG, Samsung, Vodafone and T-Mobile.

These licenses are individually negotiated, typically involve payments of tens of millions of dollars a year, and allow companies to modify code and not make those changes public, said a lawyer involved in rounds of these negotiations, who asked not to be named because the contracts were private.

Google took a different course in the cellphone business. In 2007, it founded the Open Handset Alliance and was joined by several cellphone makers and telecommunications companies. Its Android software is open source, un-

der a different licensing model, and outside Oracle's control.

"The dispute between Oracle and Google is really about control -- Google's ability to control the evolution of the Android technology," said John Rizzo, vice president for technology strategy for the Aplix Corporation, which makes Java-based software tools. Aplix is a member of both the Open Handset Alliance and the Java Community Process.

Still, Google and Sun held talks over the last three years about reaching an agreement, but they made little progress. Sun prepared the basis of the lawsuit that Oracle eventually filed, and identified the seven patents that Oracle accuses Google of infringing, said a former Sun manager, who asked not to be identified because of the suit.

Sun eventually chose not to sue Google, the former Sun manager said, because it decided a patent lawsuit would undercut the company's open-source efforts under Jonathan Schwartz, Sun's chief executive who resigned in February after the Oracle acquisition.

In 2005, Sun released an open-source version of its Solaris operating system, which is used in data centers. Software patents are controversial, especially among open-source developers. If Sun filed a patent suit, the former manager said, Mr. Schwartz feared the move would alienate many open-source enthusiasts and potential customers, from Silicon Valley start-ups to governments around the world that are pursuing open-source initiatives.

The legal preparations were led by Noreen Krall, Sun's former chief intellectual property counsel. Ms. Krall joined Apple this year as its senior director for intellectual property law and litigation. In March, Apple sued the cellphone maker HTC, saying its Android-based phone infringed on patents for Apple's iPhone. Google was not sued, but Google issued a statement saying "we stand behind" the Android operating system and its industry partners.

With its purchase of Sun, "Oracle acquired a lawsuit it could bring," said Eben Moglen, a law professor at Columbia who advises on free and open-source software projects. And Oracle's chief executive, Lawrence J. Ellison, Professor Moglen added, is "taking advantage of this asset at a time when others are interested in fighting Android." Mr. Ellison is a close friend of Apple's chief executive, Steven P. Jobs.

Oracle is mainly a traditional commercial software company, making its money selling software licenses. Oracle supports Linux, but as a way to reduce the total hardware and software costs to customers using its database software, the company's profit-making jewel. Oracle is pulling back from OpenSolaris.

For its part, Google is not in the traditional software business. Its model mimics broadcast television -- its services (including software) are free, and it makes money on advertising.

The noncombatants -- open-source developers -- are hoping for an armistice between Oracle and Google, or at

least a swift resolution to remove the uncertainty.

"It's really hard to predict the consequences of big companies being nasty to each other," said Mr. Lea, the university computer scientist.

PHOTO: Andy Rubin oversees product strategy for the Android operating system at Google. (PHOTOGRAPH BY WALLY SANTANA/ASSOCIATED PRESS) (B4)

---- INDEX REFERENCES ---

COMPANY: SUN MICROSYSTEMS BELGIUM NV; SUN MICROSYSTEMS DO BRASIL INDUSTRIA E COMERCIO LDA; VODAFONE EGYPT TELECOMMUNICATIONS COMPANY S AE; SUN MICROSYSTEMS DANMARK APS; SOFTWARE AG; SUN MICROSYSTEMS OY; HEWLETT PACKARD EUROPE BV; SUN MICROSYSTEMS SLOVAKIA S R O; VODAFONE AG; SUN CORP; HEWLETT PACKARD EUROPA HOLDING BV; SUN MICROSYSTEMS POLAND SP ZOO; ORACLE AMERICA INC; ORACLE NORGE AS; SUN MICROSYSTEMS SCOTLAND BV; HEWLETT PACKARD FINANCIAL SERVICES CO; HEWLETT PACKARD HOLDING FRANCE SAS; VODAFONE LTD; SUN MICROSYSTEMS DE CHILE SA; SUN MICROSYSTEMS SCOTLAND LP; NOKIA PTE LTD; HEWLETT PACKARD INDIGO BV; SUN MICROSYSTEMS ITALIA SPA; GOOGLE SWEDEN AB; LAFARGE SA ADR; ORACLE DANMARK APS; HEWLETT PACKARD SIA; HEWLETT PACKARD S R O; HEWLETT PACKARD; APLIX INC; HEWLETT PACKARD CO; VODAFONE D2 GMBH; SUN MICROSYSTEMS AUSTRALIA PTY LTD; HEWLETT PACKARD INDIA SALES PVT LTD; SUN MICROSYSTEMS (HELLAS) SA; SUN MICROSYSTEMS PTE LTD; JAVA; HEWLETT PACKARD INTERNATIONAL BANK PLC; APPLE GMBH; SUN MICROSYSTEMS LTD; SUN MICROSYSTEMS IRELAND LTD; HEWLETT PACKARD GOUDA BV; HEWLETT PACKARD TEKNOLOJI COZUMLERI LTD STI; HEWLETT PACKARD PHILIPPINES CORP; SUN MICROSYSTEMS LLC; HEWLETT PACKARD GESELLSCHAFT MBH; SUN MICROSYSTEMS; SUN MICROSYSTEMS AS; SUN MICROSYSTEMS SCOTLAND HOLDING LP; HEWLETT PACKARD D O O DRUZBA ZA TEHNOLOSKE RESITVE; HEWLETT PACKARD (ROMANIA) SRL; SUN MICROSYSTEMS (NZ) LTD; SAMSUNG; NOKIA CORP; ORACLE SVENSKA AB; MICROSOFT DEVELOPMENT CENTER COPENHAGEN APS; SUN MICROSYSTEMS AB; SUN MICROSYSTEMS GLOBAL SERVICES BV; SUN MICROSYSTEMS OF CANADA INC; SUN MICROSYSTEMS CZECH S R O; HEWLETT PACKARD SOUTH AFRICA (PROPRIETARY) LTD; GUANGXI LIUGONG MACHINERY CO LTD; SUN MICROSYSTEMS (SCHWEIZ) AG; HEWLETT PACKARD APS; HEWLETT PACKARD NORGE AS; SUN MICROSYSTEMS EXCHANGE INC; GOOGLE LIMITED LIABILITY COMPANY GOOGLE OOO; HEWLETT PACKARD NEW ZEALAND; SUN MICROSYSTEMS KK; LG CORP; VODAFONE HUNGARY MOBILE TELECOMMUNICATIONS CO LTD; HEWLETT PACKARD DEVELOPMENT COMPANY LP; HEWLETT PACKARD FRANCE SAS; NOVADIGM INC; SUN MICROSYSTEMS SOUTH AFRICA; LEAGUE ALLOY CO LTD; HEWLETT PACKARD INDUSTRIAL PRINTING SOLUTIONS EUROPE BVBA; HEWLETT PACKARD PORTUGAL LDA; ORACLE CORP; NOKIA KOMUNIKASYON AS; ORACLE; GOOGLE DENMARK APS; JAVA BHD; HEWLETT PACKARD MANUFACTURING LTD; SUN MICROSYSTEMS EGYPT LLC; SUN MICROSYSTEMS SCOTLAND LTD; HEWLETT PACKARD LTD; HEWLETT PACKARD TECHNOLOGY LICENSES AND LICENSING LTD; HEWLETT PACKARD DOO; OBSHCHESTVO S OGRANICHENNOI OTVETSTVENNOSTIU "NOKIA"; GOOGLE SPAIN SL; HEWLETT PACKARD OY; SUN MICROSYSTEMS (MIDDLE EAST) BV; GOOGLE CZECH REPUBLIC S R O;

SUN MICROSYSTEMS EUROPE PROPERTIES BV; HEWLETT PACKARD GMBH; VODAFONE TELEKOMUNIKASYON AS; APLIX CORP; MICROSOFT CORP; SUN MICROSYSTEMS BILGISAYAR SISTEMLERI LTD STI; SUN MICROSYSTEMS MIDDLE EAST; APPLE RETAIL ITALIA SRL; SUN MICROSYSTEMS GES MBH; SUN MICROSYSTEMS NEDERLAND BV; HEWLETT PACKARD SARL; GOOGLE INC; SML SOCIETE MARSEILLAISE DE LINGERIE SARL; VODAFONE CZECH REPUBLIC AS; HEWLETT PACKARD NEDERLAND BV; SUN MICROSYSTEMS EUROPEAN HOLDING BV; SUN MICROSYSTEMS GMBH; SAMSUNG SEMICONDUCTOR INC

NEWS SUBJECT: (Information Technology Crime (1IN42); Legal (1LE33); Intellectual Property (1IN75); Patents & Trademarks (1PA79); Economics & Trade (1EC26); Business Litigation (1BU04); Corporate Events (1CR05); Technology Law (1TE30); Online Legal Issues (1ON39); Business Management (1BU42); Business Lawsuits & Settlements (1BU19))

INDUSTRY: (Software Products (1SO56); Electronics (1EL16); I.T. (1IT96); Java (1JA10); Software (1SO30); Palmtop Computing (1PA77); Mobile Phones & Pagers (1WI07); Entertainment (1EN08); Information Management (1IN35); Internet Regulatory (1IN49); Communications Software (1CO45); Palmtop Operating Systems & Software (1PA15); Database Management Systems (1DA88); Broadcast TV (1BR25); Software Development (1SO24); Consumer Products & Services (1CO62); Internet (1IN27); Consumer Electronics (1CO61); Internet Software (1IN50); Software Technology (1SO75); Telecom (1TE27); Internet Technology (1IN39); I.T. in Telecom (1IT42); Telecom Equipment (1TE98); Security (1SE29); Programming Languages (1PR09); Telecom Consumer Equipment (1TE03); TV (1TV19))

Language: EN

OTHER INDEXING: (ANDROID; APACHE WEB; APLIX; APLIX CORP; APPLE; GOOGLE; HEWLETT PACKARD; JAVA; JAVA COMMUNITY PROCESS; LG; LINUX; MICROSOFT; NOKIA; OPEN HANDSET ALLIANCE; ORACLE; SAMSUNG; SOFTWARE; SOFTWARE WAR PITS; STATE UNIVERSITY; SUN; SUN MICROSYSTEMS; VODAFONE) (Douglas Lea; Eben Moglen; Ellison; I.B.M., Oracle; John Rizzo; Jonathan Schwartz; Kent Walker; Krall; Lawrence J. Ellison; Lea; Licensees; Moglen; Noreen Krall; Oracle; Schwartz; Sharing; Steven P. Jobs.; WALLY SANTANA)

EDITION: Late Edition - Final

Word Count: 1279
8/30/10 NYT B1
END OF DOCUMENT

EXHIBIT K

Google Android, Video Games Dominate Mobile World Congress

Published February 21, 2011 | Reuters

The 2011 installment of Mobile World Congress, which gathered 60,000 professionals from 200 countries, served as the official coming-out party for the next generation of Google Android smartphones and tablets.

Android had a massive, two-floor booth packed with all the new smartphones and tablets shipping in the coming months running off the various Android platforms, including Sony Ericsson's Xperia Play phone, which operates on Gingerbread, and LG's Optimus Pad tablet, which is powered by Honeycomb.

The Android booth featured a huge arcade section where developers from around the world showcased the next generation of mobile games. While graphics and gameplay have improved in the mobile space over the years, the new wave of smartphones and tablets will push the capabilities of these devices to the level of PlayStation 3 or Xbox 360 -- and beyond even the best iPhone 4 and iPad games that are currently on the market.

Check out FOXBusiness.com's new technology page at foxbusiness.com/technology

"Video games drive the mobile business because they're the best showcase of what these new devices can really do," said Gonzague de Vallois, senior vice president of publishing at Gameloft, one of the largest mobile game companies in the world. "Our experience in the console market is helping us deliver quality titles to these new devices for the mobile market."

Gameloft, which previously supported Apple's launch of iPhone and iPad with games, had franchises like Asphalt 6, Let's Golf and NOVA running in autostereoscopic (glasses free) 3D at LG's booth on the new Optimus 3D phone. The company also will support the March launch of Xperia Play with 10 titles, including Star Battalion. And the publisher has recently started developing new games to run on NVIDIA's Tegra 2 dual-core technology.

Tegra 2 brings PC gaming graphics and speed from just a few years ago to the mobile space on new Android smartphones and tablets, which will run 10x faster than the processors in today's smartphones like iPhone 4. The Tegra 2's dedicated graphics processor also delivers 1080p HDTV playback of movies, TV shows and games. On battery life, Tegra's ultra low-power design delivers over 16 hours of HD video or 140 hours of music on a single charge.

"This technology will impact the games we release because it will absolutely increase the available market," said Mike Breslin, vice president of marketing, Glu Mobile. "There will be a lot more people with access to these new smartphones and tablets thanks to the marketing push from Google, NVIDIA and all of the consumer electronics companies, carriers and headset makers."

Last fall, Android overtook Symbian to become the top smartphone platform in the world. According to research firm Canalys, global sales of Android phones in the fourth quarter of 2010 was 33.3 million, compared to 31 million Symbian phones, 16.2 million Apple phones, 14.6 million RIM devices and 3.1 million Microsoft phones.

Overall, the global smartphone market grew 89% compared to the fourth quarter of 2009, exceeding 100 million units for the first time. And 2011 is poised to be an even bigger breakthrough year for Android.

Another new trend showcased at Mobile World Congress was cross-platform gaming on Android and Tegra devices, which will allow players to team up in games like Trendy Entertainment's Dungeon Defenders: First Strike to play across PlayStation 3, PC and mobile platforms.

"This is a really profound change in the way mobile games are designed," said Jeremy Stieglitz, development director at Trendy Entertainment. "There will be a huge influx of quality content very quickly on these new Tegra-based platforms, where you basically can have the same game running on a mobile device as you have on a PC or console."

NVIDIA showcased brand new Tegra 3 technology at the show, code-named Kal-El after Superman. New phones will ship with this quad-core chip beginning this August and tablets will follow in late fall. NVIDIA demonstrated a game, Great Battles Medieval, that ran at 720p HD and featured 650 enemy soldiers on the field at once.

According to Michael Rayfield, general manager of NVIDIA's mobile business unit, over the next three years projects codenamed

Wayne, Logan, and Stark will further push the gaming potential of mobile devices. By 2014, when Stark becomes a reality, the technology will feature a 75x improvement over the performance of today's Tegra 2.

"Our customers and partners have already indicated that they're confident they can use everything we give them," said Rayfield.

HTC introduced a new tablet, Flyer, which will introduce streaming subscription video game service OnLive to the mobile space. Gamers will be able to play games like Assassin's Creed Brotherhood, NBA 2K11 and LEGO Harry Potter on the new tablet, or connect the tablet to any HDTV, without needing to buy any new hardware or software.

There is a negative side to the increased capabilities of mobile gaming. And that's for portable gaming companies like Sony, which will launch its Next Generation PlayStation (NGP) system this fall.

"IHS believes that the market opportunity for a specialist device such as the NGP is shrinking rather than growing, and that short- and medium-term market conditions are less supportive of the release of a high-end handheld console," said Piers Harding-Rolls, video game analyst at HIS Screen Digest.

Harding-Rolls said by the end of the fourth year after its release, the NGP is expected to accrue a total installed base of 22.8 million units. In comparison, the PSP achieved a base of 30.7 million, 34.8% higher, during the same length of time.

"The competitive landscape for handheld and on-the-move gaming has been highly disrupted in recent times, with disruption occurring on the device, content and distribution levels," added Harding-Rolls.

Moving forward, more consumers will use one device for everything, including gaming, multimedia, entertainment and Web browsing. And Android is perfectly positioned to take advantage of this shift.

EXHIBIT L

THE WALL STREET JOURNAL.
WSJ.com

TECHNOLOGY | FEBRUARY 17, 2011

Google's Android the Talk of Barcelona

System Powers Slew of New Devices; Some Fear Market Duopoly With Apple's iOS

By SHAYNDI RAICE

A year after wireless carriers gave Google Inc. a testy reception at their big industry conference in Barcelona, the software company's Android operating system has become the star of the show.

Android powers every significant device launched at the Mobile World Congress and is benefiting from a big marketing push by Google to trumpet its arrival.

Beyond the hype, it's becoming increasingly evident to carriers, handset makers and app developers that Android is on equal footing, if not surpassing in some ways, Apple Inc.'s iOS platform. The concern now is that it could become too powerful.

Demotix

Attendees mingle at this week's conference in Barcelona, where many companies launched devices powered by Google's Android operating system.

Nokia Corp. Chief Executive Stephen Elop said one reason he decided to put his chips on Microsoft Corp.'s Windows Phone 7 platform instead of Android was because he was concerned about creating a duopoly in the marketplace where Google and Apple would maintain all the power.

That underscores the ambivalence of carriers that are relying on Android for hit devices but worry about market concentration.

"To just have two companies is probably not healthy for the ecosystem," said Fared Adib, vice president for device operations at Sprint Nextel Corp.

Google's Android chief, Andy Rubin, puts no stock in that claim. "It makes me uncomfortable when people say something open source is too powerful," said Mr. Rubin, referring to the fact that Android code is available to anyone who wants to use it without any fee. "One of the reasons we've achieved such adoption is because we've removed all control."

Journal Community ›

Carriers have been wary of Google for some time. A year ago in Barcelona, Vodafone Group PLC Chief Executive Vittorio Colao blasted Google's dominance of mobile advertising, suggesting regulators should take a look. Telefonica SA Chairman Cesar Alierta chimed in, complaining that search engines ride on top of carriers' networks and don't share the money they make doing so.

None of that has damped Android fever this year in Barcelona.

The little green Android robot is ubiquitous. There are Android pins, an Android slide and there was even an Android drink at an exclusive party held the by company last night. Twitter Inc. Chief Executive Dick Costolo enjoyed some champagne from flutes sporting an Android logo in the VIP area with Google Chairman Eric Schmidt, as hordes of people waited outside the club clamoring for entrance to the show's most talked-about party.

Samsung Electronics Co. introduced a new Galaxy S smartphone and tablet. HTC Corp. announced its first tablet along with five new smartphones. LG Electronics Inc. showed off its 8.9-inch G-Slate tablet and the first 3D phone. Sony Ericsson introduced the Xperia Play. All of those devices run on Android.

Google's big presence at what has traditionally been a carriers' conference highlights the all-important role of software in a mobile industry where smartphones and data use are creating most of the value.

Android devices have been around for a couple of years but only really started getting traction in the past 15 months or so with the introduction of high-end devices by Motorola Mobility Holdings Inc. and HTC. Google says it's now activating over 300,000 Android-powered devices a day, with 170 devices on the market and 169 carriers.

Some companies feel that even if Google is gaining uncomfortable amounts of power in the mobile market, they have no choice but to get on board. LG, Samsung and especially Motorola all leveraged Android to get traction in the smartphone market.

"I don't know that you can have it all," said Alain Mutricy, senior vice president, portfolio and devices product management for Motorola Mobility. "If you want to be at the forefront of technology, that's what you have to do."

The bigger concern for carriers this year may be Apple. The combination of hugely popular devices with a closed ecosystem of applications and billing gives the company a tenacious hold on the loyalty of carriers' customers.

This year, Vodafone's chief, Mr. Colao, specifically criticized vertically integrated closed systems when it comes to in-application billing and called for openness and competition.

AT&T Inc. Chief Executive Randall Stephenson said content should be "device agnostic" and stored in the "cloud" on AT&T's network.

Mr. Schmidt picked up the theme in his keynote Tuesday evening, saying cloud-based applications built on HTML5 could allow devices to become interchangeable.

"All of a sudden, you have this ability, this agility," he said.

—Gustav Sandstrom contributed to this article.

EXHIBIT M

MONDAY NOV 05, 2007

Congratulations Google, Red Hat and the Java Community!

I just wanted to add my voice to the chorus of others from Sun in offering my heartfelt congratulations to Google on the announcement of their new Java/Linux phone platform, Android. Congratulations!

I'd also like Sun to be the first platform software company to commit to a complete developer environment around the platform, as we throw Sun's NetBeans developer platform for mobile devices behind the effort. We've obviously done a ton of work to support developers on all Java based platforms, and were pleased to add Google's Android to the list.

The Java platform has come a long way - we're on the vast majority of mobile devices in the marketplace today (well **over a billion phones** at last count), and with Google's mobile services (like gMail and Google Maps), along with Yahoo!'s Go Mobile, alongside a massive spectrum of incredible entertainment offerings from folks like Electronic Arts, we have by far and away the most complete content ecosystem on the market today. Enabling carriers, handset manufacturers, content creators - and most of all, consumers - to get the most from their mobile devices.

And needless to say, Google and the Open Handset Alliance just strapped another set of rockets to the community's momentum - and to the vision defining opportunity across our (and other) planets.

Today is an incredible day for the open source community, and a massive endorsement of two of the industry's most prolific free software communities, Java and Linux.

Stay tuned for specific details - for those so inclined, download NetBeans here to check out the industry's most popular IDE for mobile Java development.

And in the spirit of offering congratulations, I'd like to offer a similar shout out to our friends at Red Hat Linux - who today announced their support for the OpenJDK project. With friends like Google and Red Hat, it sure seems like the momentum behind Java's on the rise...

Posted on 01:27PM Nov 05, 2007 | Comments[76]

Comments:

Why isn't Sun part of the OpenHandsetAlliance? Is it another case of NIH, since Google is not using JavaFXMobile so Sun doesn't want to be part of it?

Posted by **Enaiel** on November 05, 2007 at 02:07 PM PST #

Jonathan, it is amazing the amount of value this has brought to Google's share prices as the investment world was speculating on Google's play in this industry.

It is companies like Sun and Google, which are helping the possibility and connectivity of the net expand on a daily basis.

Thank you

EXHIBIT N

SUN TECHNICAL REFERENCE LIBRARY

The NeWS Book

An Introduction to the Network/extensible Window System

SPRINGER-VERLAG

James Gosling
David S.H. Rosenthal
Michelle J. Arden
Sun Microsystems, Inc.
Mountain View, CA 94043, USA

Graphics Designer and Editor: David A. LaVallée

Library of Congress Cataloging-in-Publication Data
Gosling, James
The NeWS book : an introduction to the Networked Extensible Window System / James Gosling, David S.H. Rosenthal, Michelle J. Arden.
p. cm.—(The Sun technical reference library)
Bibliography: p.
Includes index.
ISBN 0-387-96915-2
1. Windows (Computer programs) 2. NeWS (Computer program)
I. Rosenthal, David S.H. II. Arden, Michelle J. III. Title.
IV. Series.
QA76.76.W56A73 1989
005.4'3—dc20 89-11329

Camera-ready copy prepared by the authors.
Printed and bound by R.R. Donnelley & Sons, Harrisonburg, Virginia.
Printed in the United States of America.

9 8 7 6 5 4 3 2 1 Printed on acid free paper.

ISBN 0-387-96915-2 Springer-Verlag New York Berlin Heidelberg
ISBN 3-540-96915-2 Springer-Verlag Berlin Heidelberg New York

complex curves. Andrew, X10 and X11 lack these capabilities, or rather, they leave their implementation to the developer.

The NeWS graphics model is based on the stencil/paint model provided by the PostScript language. This graphics, or imaging model, is at a high enough level of abstraction to provide device independence along with a rich set of graphics capabilities to NeWS-based applications. Applications are not written in terms of specific hardware, therefore they need not be concerned about the resolution of the display, or whether the display is monochrome or color. Also, NeWS clients can automatically benefit from special high-performance graphics hardware, since the imaging model maps easily to many graphics accelerators. System vendors can provide acceleration through their NeWS server implementations while keeping the NeWS programming and graphics interfaces constant.

NeWS applications are even isolated from whether the output device is a printer or a display. Since NeWS contains a PostScript language interpreter similar to the one found in laser printers, a given series of PostScript language statements will render the same image whether sent to a NeWS window or to a printer containing a PostScript language interpreter. Thus, it is easy to preview printer output on the screen, or to send the contents of a window to the printer.

The two images in Figure 2.5 demonstrate the device-independent nature of the PostScript language. The image on the left was printed by sending a PostScript program directly to the printer; the one on the left is a snapshot of the same PostScript program rendered within a NeWS window.

2.6 Networking

In a distributed networked environment, accessing windows on another machine should be as natural as transparently accessing remote files via Sun's Network File System (NFS). Workstations and, increasingly, personal computers, are best used as elements in a heterogeneous environment, communicating over a network with other machines ranging from low-cost terminals, through workstations, to supercomputers. NeWS puts the resources of such a distributed computing environment on the screen. NeWS client programs don't have to run on the computer with the screen; they may be distributed in different ways across both client and server machines depending upon the resources available and the usage of the network. Experiences with Andrew and X indicate that the flexibility of client program location is valuable both for good local performance and an efficient use of resources across the global network.

Real-time response over a network is difficult in a server-based window system since the server usually has to pass messages to the client and wait for a response from the client whenever input or output occurs requiring

EXHIBIT O

Re: EOLAS ACQUIRES MILESTONE INTERNET SOFTWARE PATENT

From: James Gosling <jag@scndprsn.eng.sun.com>
Date: Mon, 21 Aug 1995 14:48:43 +0800
Message-Id: <9508212148.AA18759@norquay.Eng.Sun.COM>
To: info@eolas.com, ses@tipper.oit.unc.edu
Cc: www-talk@w3.org

```
> > The licensed technology was invented in 1993 by a team led by Eolas CEO, Dr.
> > Michael Doyle, a UCSF faculty member and past Director of the university's
>
> In that case, the patent is worthless. Dynamic exchange of executable
> code over the internet for use in rendering documents has been addressed
> many times before. For example, a standard for the exchange of such
> information was discussed at the spring 1992 meeting of the implementors
> working group of NISO sub-committee Z39.50 which took place at the
> Library of Congress in Washington.

There's also Java which was started in 1990 and first demo'd in 1991;
the NeWS window system which transmitted executable application code
starting in 1985; PostScript was really dynamic exchange of executable
code for rendering documents, and it was done in 1980.  Based of course
on ideas from Interpress where executable code for rendering documents
was being transmitted in the late 70s.  Then there was a lisp system,
whose name a forget, which ran on Xerox's PDP-10 clones (MAXC) and
could dynamically download ui's/behaviour/graphics to Alto's in the
early 70s.
```

Received on Monday, 21 August 1995 17:49:00 GMT

EXHIBIT P

MICROSOFT PRESS®

COMPUTER DICTIONARY

SECOND EDITION

THE COMPREHENSIVE STANDARD FOR BUSINESS, SCHOOL, LIBRARY, AND HOME

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Cataloging-in-Publication Data

Microsoft Press computer dictionary : the comprehensive standard for business, school, library, and home / Microsoft Press. -- 2nd ed.
p. cm.
ISBN 1-55615-597-2
1. Computers--Dictionaries. 2. Microcomputers--Dictionaries.
I. Microsoft Press. II. Title: Computer dictionary.
QA76.15.M54 1993
004'.03--dc20 93-29868
CIP

Printed and bound in the United States of America.

2 3 4 5 6 7 8 9 MLML 9 8 7 6 5 4

Distributed to the book trade in Canada by Macmillan of Canada, a division of Canada Publishing Corporation.

Distributed to the book trade outside the United States and Canada by Penguin Books Ltd.

Penguin Books Ltd., Harmondsworth, Middlesex, England
Penguin Books Australia Ltd., Ringwood, Victoria, Australia
Penguin Books N.Z. Ltd., 182-190 Wairau Road, Auckland 10, New Zealand

British Cataloging-in-Publication Data available.

Project Editor: Casey D. Doyle
Manuscript Editor: Alice Copp Smith
Technical Editors: Mary DeJong, Jeff Carey, Dail Magee, Jr., Jim Fuchs, Seth McEvoy

term is usually limited to describing a system with two microprocessors; a system with a microprocessor and a math coprocessor is not considered a dyadic system. In mathematics, a dyadic operation is one in which there are two operands. In Boolean algebra, a dyadic Boolean operation is, again, one in which there are two operands, both of which are significant. Dyadic Boolean operations are those such as AND and OR in which the outcome depends on both values. Such operations are commonly used to create truth tables. *Compare* unary; *see also* Boolean algebra, operand.

dye-polymer recording A type of recording technology used with optical discs in which dye embedded in a plastic polymer coating on an optical disc is used to create minute bumps on the surface that can be read by a laser. Dye-polymer bumps can be flattened and re-created, thus making an optical disc rewritable, as opposed to being recordable only once.

dynamic An adjective used to describe events or processes that occur immediately and concurrently as opposed to those planned for in advance or reacted to after the fact. *Dynamic* is used in reference to both hardware and software; in each case it describes some action or event that occurs when and as needed. In nondynamic memory management, a program is given a certain amount of memory when the program is first run and must run within that constraint. In dynamic memory management, a program is able to negotiate with the operating system when it needs more memory.

dynamic address translation Abbreviated DAT. On-the-fly conversion of memory-location references from relative addresses ("three units from the beginning of X") to absolute address ("location number 123") when a program is run. Dynamic address translation depends on conditions existing within the system at the runtime of a program; for example, it might depend on exactly where in memory a particular part of a program is loaded by the operating system.

dynamic allocation The allocation of memory during program execution according to current needs. Dynamic allocation almost always implies that dynamic deallocation is possible too, so data structures can be created and destroyed as required. *Compare* static allocation; *see also* allocate, deallocate.

dynamic binding Also called late binding. Binding (converting symbolic addresses in the program to storage-related addresses) that occurs during program execution. The term often refers to object-oriented applications that determine, during runtime, which software routines to call for particular data objects. For example, an application might define a class named "artwork," with subclasses for paintings, sculptures, ceramics, and so on. Each of these classes would have a routine named "dollarvaluenow" that would calculate the current value of a piece of art, based in part on the class's unique characteristics and also on the state of the market for art. Given an artwork object, dynamic binding would ensure that the correct "dollarvaluenow" routine was called to compute the current value. *Compare* static binding.

Dynamic Data Exchange Abbreviated DDE. A form of interprocess communication (IPC) implemented in Microsoft Windows and OS/2. When two or more programs that support DDE are running simultaneously, they can exchange information and commands. For example, a spreadsheet with a DDE link to a communications program might be capable of keeping stock prices that are displayed in the spreadsheet current with trading information received over the communications channel. *See also* interprocess communication.

dynamic dump A listing, either stored on disk or sent to a printer, of memory contents generated at the time of a break in the execution of a program; a useful tool for programmers interested in knowing what is happening at a certain point in the execution of a program.

dynamic-link library A feature of the Microsoft Windows family of operating systems and the OS/2 operating system that allows executable routines—generally serving a specific function or set of functions—to be stored separately as files with DLL extensions and to be loaded only when

(or main) and secondary (or auxiliary) storage devices. When this distinction is made, the primary storage device is the computer's random access memory (RAM)—impermanent, but a storage device nevertheless, however temporary its contents. The secondary storage includes the computer's more permanent storage devices, such as disk and tape drives.

storage location The position at which a particular item can be found. A storage location can be an addressed (uniquely numbered) location in memory or it can be a uniquely identified location on disk, tape, or a similar medium—for example, a particular side, track, and sector on a disk.

storage media The various types of physical material on which data bits are written and stored. Common storage media for computer data are floppy disks, hard disks, tape, optical discs, and (for output only) paper.

storage tube *See* direct view storage tube.

store-and-forward A message-passing technique used on communications networks in which a message is held temporarily at a "collecting" station between the sender and receiver before being forwarded to its intended destination. Store-and-forward message routing can take longer than communication via direct, physical connections, but it offers several advantages, especially over large networks separated by long distances: It minimizes or eliminates "traffic jams" and thus contributes to effective use of communications lines; it allows messages to be sent to machines or networks even when they are not on line; and it enables transmission during off hours, when traffic or costs are lowest.

stored program concept The underlying concept of most system architectures today, credited largely (although not exclusively) to John von Neumann. The concept is that both programs and data are in direct access storage (random access memory, or RAM), allowing code and data to be treated interchangeably (including modifications to both) and avoiding the timing problems involved when code is located on a sequential storage medium (as was the case in many early computers). *See also* von Neumann architecture.

straight-line code Program code that follows a direct sequence of statements, rather than skipping ahead or jumping back via transfer statements such as GOTO, JUMP, and so on. *Compare* spaghetti code; *see also* GOTO statement.

streaming tape *See* tape.

stream-oriented file A file used to store a more-or-less-continuous series of bits, bytes, or other small, structurally uniform units.

strikethrough One or more lines drawn through a selected range of text. Marking text in this way indicates that the text is to be deleted at some future time. A strikethrough is implemented on printers in different ways: Text-based printers overstrike hyphens on the text; graphic printers, such as laser printers, can actually draw a line through the text.

string A data structure composed of a sequence of characters, usually representing human-readable text. A string's current length (the actual sequence stored) might not be the same as its allotted maximum size; consequently, some languages provide a way of determining current length, usually by using a delimiting character at the end or by counting the number of characters.

string variable An arbitrary name assigned by the programmer to a string of alphanumeric characters; after making the assignment, the programmer can use or change the contents of the string by simply referencing the string variable name. Not all programming languages support string variables. *See also* string.

strobe A timing signal that initiates and coordinates the passage of data, typically through an input/output (I/O) device interface, such as a keyboard or printer port.

stroke In data entry, a keystroke—a signal to the computer that a key has been pressed; in typography, a line representing part of a letter; in paint programs, a "swipe" of the paintbrush made with mouse or keyboard in creating a graphic; in display technology, a line created as a vector (a path between two coordinates) on a vector graphics display (as opposed to a line of pixels drawn dot by dot on a raster graphics display).

stroke font A font printed by drawing a combina-

EXHIBIT Q

Compiler Construction for Digital Computers

DAVID GRIES

WILEY INTERNATIONAL EDITION

Library of Congress Catalogue Card Number: 74-152496

Printed in the United States of America.

19 18

Chapter 11.
Internal Forms of the Source Program

If space is a problem,
the project are com
translate into an int
mechanically. In
essentially in the ord
is a big help for
Actually these inter
interpreting. That
execute the source pro
form.

This chapter introdu
internal forms. The
be shown in each to pr
course, that the parti
language and the aims
DIMENSION statements
program, since all the
no code need be ge
declaration INTEGER K
the internal forms, si

One must also deter
should be. Should it
one type to another, c
example, a for-loop
just assignment, go to
appear on a much high
In general, the origin
a more detailed, tr
take place and in gene

We begin our discus
operands, and how t
describe our first int
discuss quadruples
triples (section 4), a
(section 5).

The reader should re
forms exclusively.
used, depending on the
designer.

```
BEGIN INTEGER K;
      ARRAY A[1:
      K := 0;
L:    IF I > J
      THEN K :=
      ELSE BEGI
END
```

FIGUR

If space is a problem, or if the source language or the goals of the project are complicated enough, a compiler will first translate into an internal form which is easier to handle mechanically. In most internal forms, operators appear essentially in the order in which they are to be executed; this is a big help for subsequent analysis and code generation. Actually these internal forms could also be used for interpreting. That is, we could write a program which would execute the source program as it is represented in its internal form.

This chapter introduces several of the more frequently used internal forms. The ALGOL program segment of Figure 11.1 will be shown in each to provide examples. One should realize, of course, that the particular representation depends on the source language and the aims of the compiler. For example, in FORTRAN, DIMENSION statements need not be put into the internal source program, since all the information is put into symbol tables and no code need be generated. You will also note that the declaration INTEGER K of Figure 11.1 does not appear in any of the internal forms, since no code need be generated for it.

One must also determine how detailed the initial internal form should be. Should it include operations to convert values from one type to another, or will this be done later? As another example, a for-loop could appear in its equivalent form using just assignment, go to, and conditional statements, or it could appear on a much higher level to be translated at a later time. In general, the original form is more concise and smaller, while a more detailed, translated form allows more optimization to take place and in general makes things easier.

We begin our discussion with a section on operators and operands, and how they can be represented. In section 2 we describe our first internal form, Polish notation. We then discuss quadruples (section 3), triples, trees and indirect triples (section 4), and graph representations of the program (section 5).

The reader should realize that few compilers use one of these forms exclusively. Usually, a mixture of several of them is used, depending on the whims and prejudices of the compiler designer.

```
BEGIN INTEGER K;
      ARRAY A[1:I-J];
      K := 0;
L:    IF I > J
      THEN K := K + A[I-J]*6
      ELSE BEGIN I := I+1; I := I+1; GO TO L END
END
```

FIGURE 11.1. An ALGOL Program Segment.